Implementierung und Evaluation eines hardwarebeschleunigten Load-Balancers auf einer Nvidia Bluefield DPU Architektur

---

Bachelorarbeit

von

Sten Heimbrodt

Universität Potsdam
Institut für Informatik und Computational Science
Betriebssysteme und Verteilte Systeme

Supervisor(s):
Prof. Dr. Bettina Schnor
Max Schrötter

Potsdam, 14. Juni 2025

Heimbrodt, Sten
heimbrodt@uni-potsdam.de
Implementierung und Evaluation eines hardwarebeschleunigten Load-Balancers auf einer Nvidia Bluefield DPU Architektur
Bachelorarbeit, Institut für Informatik und Computational Science
Universität Potsdam, June 2025

Ich möchte mich zunächst beim gesamten Lehrpersonal der Universität Potsdam und insbesondere bei den Mitarbeiterinnen und Mitarbeitern des Instituts für Informatik bedanken. Die Jahre meines Studiums in Potsdam waren für mich äußerst prägend. Mein besonderer Dank gilt Frau Prof. Dr. Schnor, die mir die Möglichkeit gegeben hat, meine Abschlussarbeit in einem für mich hochinteressanten Thema zu verfassen. Ebenso danke ich Max Schrötter, der mich nicht nur fachlich, sondern auch menschlich bei der Anfertigung meiner Arbeit unterstützt hat. Ein großes Dankeschön geht auch an meine Familie, insbesondere an meine Mutter. Sie stand stets hinter mir, selbst in Zeiten, in denen mein Weg noch nicht klar erkennbar war. Die letzten Worte dieser Danksagung möchte ich jedoch meiner Lebensgefährtin widmen. Sie war mir nicht nur in schwierigen Momenten eine Stütze, sondern ist auch eine wesentliche Quelle meiner Inspiration im Leben. Ohne sie wäre ich nicht hier.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Potsdam, den 14. Juni 2025

Sten Heimbrodt

Zusammenfassung

Lastverteilung ist eines der zentralsten Probleme der Informatik überhaupt. So ist es von größtem Interesse, gerade vor dem Hintergrund des Klimawandels, möglichst effiziente Nutzung von Hardware-Ressourcen zu machen. Ziel dieser Arbeit ist es, einen L3/L4-Netzwerklastverteiler auf einer SmartNIC-Architektur zu implementieren und hinsichtlich seiner Leistungsfähigkeit zu untersuchen. Außerdem werden die zur Entwicklung verwendeten Bibliotheken analysiert und eine Übersicht geschaffen, wie etwaige Programmbibliotheken verwendet werden, um eine DPU der BlueField-Reihe zu programmieren. Im Rahmen dieser Arbeit wurde XenoFlow entwickelt. XenoFlow ist ein Lastverteiler, der vollständig auf der spezialisierten Hardware der BlueField ausgeführt wird. Der Vergleich zu anderen Lastverteilern konnte zeigen, dass eine signifikante Leistungssteigerung erfolgen kann, wenn spezialisierte Hardware zur Verarbeitung des Netzwerkverkehrs genutzt wird. Daraus lässt sich schließen, dass SmartNICs durchaus ein gutes Mittel sind, um Anwendungen, welche sich zentral mit Netzwerkverkehr beschäftigen, auf einer DPU-Karte zu implementieren. Allerdings ist ein tatsächlicher Leistungsgewinn stark von der jeweiligen Implementierung abhängig.

Inhaltsverzeichnis

1 Einführung

In modernen Rechnernetzen und Rechenzentren hat sich der Bereich der Lastverteilung zu einem der wichtigsten Teilbereiche entwickelt. Hintergrund ist, dass mit steigender Anzahl von Internetnutzenden die entsprechenden Hardwareressourcen nicht nur zur Verfügung gestellt werden müssen, sondern vielmehr eben auch die effiziente Nutzung von Interesse geworden ist.[1] Dabei gibt es verschiedene Ansätze und auch verschiedene Punkte, an denen Maßnahmen getroffen werden können, um eine breite Nutzung zu realisieren. Auch vor dem Hintergrund der globalen Erwärmung ist es sinnvoll, die Hardwarenutzung zu optimieren, da somit Energie eingespart und fossile Mittel geschont werden können. [2]

So wurden bereits relativ früh skalierbare Systeme entwickelt. Ein klassisches Beispiel ist hierbei der Einsatz von mehreren Prozessoren in Webservern der frühen 90er Jahre. Während dieser Zeit gab es technologische Vorstöße im Bereich des Internets für zivile Endanwender, wodurch dementsprechend die Nachfrage nach bereitstellender Rechentechnik stieg. Somit wurde indirekt die Zeit der Mehrkern-Prozessoren eingeläutet, obwohl diese erst Jahre später entwickelt wurden. Einen sehr ähnlichen Ansatz verfolgen die Entwickelnden moderner Lastverteiler. Dabei wird Wert auf eine horizontale Skalierung gesetzt. Es werden Ressourcen auf horizontaler Ebene nebeneinander gesetzt, um den Gesamtdurchsatz des Systems zu erhöhen. Der dem entgegenstehenden Ansatz wäre die vertikale Lastverteilung, bei der im Wesentlichen die Punktleistung eines Rechners erhöht wird. Auch die vertikale Skalierung findet breite Anwendung. Am wichtigsten hierbei ist das Hochfrequenz-Trading am Aktienmarkt sowie High-Performance-Computing im Allgemeinen.

Allerdings hat sich mit der zeitlichen Weiterentwicklung nicht nur das Interesse an Parallelisierbarkeit gesteigert, sondern ebenfalls ein Interesse an Hardwarearchitekturen, die im Gegensatz zum klassischen Prozessor nicht nur einen allgemeinen Zweck erfüllen, sondern für spezifische Anwendungsbereiche entwickelt werden. Dabei kann durch das an die Anwendung angepasste architektonische Layout dermaßen optimiert werden, dass es nicht nur zur Leistungssteigerung selbst kommt, sondern eben auch zu einer deutlich besseren Energieeffizienz. Zudem haben die letzten Vorstöße im Bereich des maschinellen Lernens dazu geführt, dass eine Menge der traditionellen Rechenzentrumsarchitekturen zwar für die modernen GPU-Cluster funktionieren, es aber eine Menge an Potenzial und Optimierungsspielraum gibt. [3] Diese Arbeit behandelt eben genau eine dieser modernen Technologien, die zwar mit einer Idee für einen Anwendungszweck ins Leben gerufen wurde, aber deren tatsächliche Anwendung noch immer hinterherhinkt. Ziel dieser Arbeit war es, einen Lastverteiler auf einer DPU-Architektur zu implementieren und zu untersuchen, wie und vor allem ob sich eine solche Plattform für diese Art von Anwendungen eignet. Außerdem wird zwischen verschiedenen Lastverteilungsansätzen verglichen.

2 Smart NICs - ein Überblick

In diesem Kapitel wird ein allgemeiner Überblick über die Architektur einer intelligenten Netzwerkkarte gegeben.

2.1 Begriffserklärung

Der Begriff **Smart Network Interface Card** (Smart NIC) ist wie allzu oft in der Computerwissenschaft mit einer Menge von Definitionen versehen. Grundsätzlich beschreibt eine Smart NIC eine Hardwareeinheit, deren Funktion die Netzwerkfähigkeit eines Rechners beschreibt. Dabei unterscheidet sie sich von den klassischen Netzwerkkarten in der Hinsicht, dass wir als Anwender beziehungsweise Entwickler Anwendungen schreiben können, die das Verhalten der Karte beeinflussen. Dabei ist nicht fest definiert, welche Funktionen gebaut werden oder überhaupt, in welchem Umfang programmiert werden kann. Nicht zuletzt ist das immerwährende Aufkommen des Begriffs Smart* dafür verantwortlich, dass sicherlich auch aus Marketinggründen ein entsprechender Name gewählt wurde. Allgemein ist eine Smart NIC eine Netzwerkkarte, die es erlaubt, Anwendungen zu erstellen, die ihr eigenes Verhalten derart verändern kann, dass Funktionen, die sonst innerhalb eines Netzwerkstapels übernommen werden würden, nun auf der Netzwerkkarte selbst ausgeführt werden können. [4]

2.2 Aufbau und Funktionsweise

Smart NICs verwenden unterschiedliche Hardwarearchitekturen, um eine Hardwarebeschleunigung zu realisieren. Allgemein gab es sogar in den 90er- und frühen 2000er-Jahren Smart NICs, die lediglich oder größtenteils klassische Prozessorarchitekturen wie RISC oder x86 verwendeten. Mit Anbruch des neuen Jahrtausends zogen jedoch immer mehr FPGA- und ASIC-basierte Architekturen in die Rechenzentren ein. [5] Das steigende Interesse an der SmartNIC Architektur ist nicht zuletzt durch den abflachenden Leistungsgewinn durch Generationsfortschritte in der Prozessorindustrie befeuert worden. Mittlerweile gilt das Moorsche Gesetz als beendet und ein Großteil der Computerkommunikation heutzutage ist ohnehin Netzwerkverkehr. [8] Meistens werden dazu TCP- und UDP-Pakete verwendet. Werden diese Pakete auf Prozessoren verarbeitet, sind die entsprechenden Kosten nicht nur hardware- sondern auch energieseitig sehr hoch. Zuletzt hat nun auch der Aufschwung der modernen Softwarearchitekturen für einen explosionsartigen Anstieg in Netzwerkverkehr gesorgt. Wurden vor zwanzig Jahren Anwendungen noch so entwickelt, dass möglichst wenig Kommunikation über ein Netzwerk läuft, wird in heutigen Systemen stark und immer mehr auf eine Microservice-basierte Architektur gesetzt. [6] Dabei wird eine komplexe Applikation in verschiedene Bereiche unterteilt, die daraufhin mittels HTTP-angebundenen

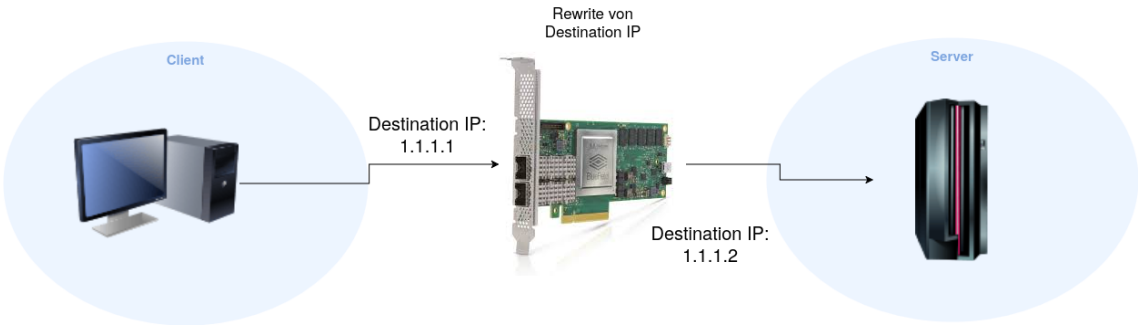

Abbildung 2.1: Einfaches Ändern einer IP-Adresse im Paket

Interfaces miteinander kommunizieren. Grund dafür ist die Idee der vereinfachten Wartbarkeit von kleinteiligen Applikationen. Zu Beginn der Entwicklung besagter Hardware war zunächst grundsätzlich die Idee, die programmierbare Funktionalität auf dem Switch, also dem Anbindungspunkt des Rechenzentrums, umzusetzen. Allerdings steigt mit der Anzahl der Knoten im Teil hinter dem Switch auch die benötigte Rechenkapazität des Switches. [7] Daher hat sich ein eher dezentraler Ansatz entwickelt, bei dem in dem Server selbst eine Karte integriert ist, auf der sich ein zusätzlicher Rechner befindet. Dabei ist die Verbindung zugunsten der größten Kompatibilität meist mittels PCI-Express umgesetzt. In der Abbildung 2.1 ist ein grundsätzliches Diagramm eines hier abstrahierten Datenpakets dargestellt. Der Client schickt ein Paket mit einer bestimmten Ziel-IP-Adresse los und zu einem gewissen Zeitpunkt erreicht das Paket die Smart NIC. Diese wendet daraufhin eine Aktion an. In diesem Fall wird die Ziel-IP-Adresse geändert und zurück in das Netzwerk geleitet. Dadurch bekommt nun ein anderer Server das Paket. Hierbei ist interessant, dass die Umleitung dem Client verborgen bleibt. Im Prinzip handelt es sich also bei dem gezeigten Beispiel um einen primitiven L3-Proxy.

2.2.1 Hardwareaufbau

SmartNICs sind in aller Regel im Grunde völlig eigenständige Systeme mit Prozessoreinheit, Arbeitsspeicher und teilweise sogar eigenständiger Netzwerkein- und -ausgabe über einen separaten Hardwareport. Die Idee dabei ist es, ein System zu schaffen, auf dem eigenständig Applikationen ausgeführt, aber auch Dienste wie Docker oder Kubernetes angewendet werden können. So wird auch die Verbindung zur Entwicklung über etwaige Dienste wie SSH hergestellt. Allerdings bieten auch einige Hersteller eigenständige Lösungen an, welche in aller Regel mittels **Application Programming Interfaces** kurz APIs zur Verfügung gestellt werden. Hauptgrund sind allerdings, wie so oft in solch hochkritischen Systemen, Sicherheitsbedenken. Eine vollständig getrennte Einheit unterstützt in diesem Sinne das Konzept der Zero-Trust-Paradigmen. Dabei wird angenommen, dass allen Kommunikationspartnern sowie allen Beteiligten in einem System nicht zu trauen ist. Eine SmartNIC kann somit völlig autark in einem Netzwerk fungieren, ohne dass das Hostsystem selbst den Aufbau des Netzwerks oder dessen Funktionsweise kennt. [9] Die besonders im Kontext der Netzwerkverarbeitung relevanten Hardwareeinheiten werden mittels fest verarbeiteter Hardwareinterfaces angesprochen. Dabei werden meist die vorliegenden PCI-

Lanes als Hochgeschwindigkeitsverbindung genutzt. Und genau hier wird auch einer der größten Unterscheidungspunkte zwischen der klassischen Netzwerkpaket verarbeitenden Hardware und den modernen SmartNICs deutlich. Die meisten anderen Netzwerkadapter verwenden Mikrocontroller, um eine Konfigurierbarkeit zu gewährleisten. Die Kommunikation zwischen den Mikrocontrollern und den tatsächlichen Hardwarebeschleunigern erfolgt daraufhin meist mittels serieller Schnittstellen statt. Dabei ist natürlich das Hauptunterscheidungsmerkmal der Prozessortakt. Klassische SmartNIC **Systems on a Chip** kurz SoC verwenden Prozessoren mit mehreren Kernen, die in einem ähnlichen Bereich getaktet sind wie etablierte Server- oder Desktopchips. Dabei ist die Leistung vor allem relevant, da auch klassische Berechnungen auf den Kernen ausgeführt werden sollen. Bei den kleineren, schwächeren Mikrocontroller-Prozessoren liegen die Taktraten größtenteils klar unter 1 GHz. Diese werden allerdings auch eben nur für Konfiguration und Management verwendet und übernehmen keine direkte Aufgabe, die an der Paketverwaltung beteiligt ist. Zusammenfassend lässt sich also eine grobe Unterteilung vornehmen, bei der sich die SoC-Varianten der SmartNICs klar von den Netzwerkkarten unterscheiden, die diskrete Hardwareeinheiten für den jeweiligen Verwendungszweck besitzen und diese über einen Microcontroller verwaltet werden.

2.2.2 Path Switching

Über die letzten Jahre haben sich unter diversen anderen architektonischen Ansätzen zwei etabliert, welche für den Kontext dieser Arbeit von besonderer Bedeutung sind. Es wird hauptsächlich durch die Switchting-Funktionalität unterschieden. Die zwei entstehenden Klassen sind On-Path-Switching und Off-Path-Switching (siehe Abbildung 2.2).

On-path-SmartNICs

Bei einer On-Path-SmartNIC verhält sich die Netzwerkkarte sehr nahe an der Funktionsweise, wie man sie von einer traditionellen Netzwerkkarte erwarten würde. Pakete, die über das Kabel an die Hardware gelangen, werden zunächst von dem entsprechenden Stack der Karte verarbeitet und laufen gezwungenermaßen nicht nur über den Traffic-Manager, sondern insbesondere auch über die Kerne der SmartNIC. [10] Das heißt ein nicht unerheblicher Overhead bei der Verarbeitung des Datenverkehrs entsteht im Kernel des Betriebssystems. Es ermöglicht aber eben auch, Modifikationen an besagten Paketen vorzunehmen. Dabei steht vor allem auch die vereinfachte Entwicklung solcher Netzwerkkarten im Vordergrund. Sollte es sich wiederum um eine On-path-SmartNIC handeln, bei der Pakete direkt über den Kernel des Hostrechners verarbeitet werden, stellt sich hierbei das Problem dar, dass es so zu einer Überlastung der Anbindung zwischen SmartNIC und Host kommen kann. Außerdem liegt in solch einem Fall dann ein Großteil der Rechenlast wieder beim Host, womit die eigentliche Funktionalität der SmartNIC stark abnimmt.

Off-path-SmartNICs

Bei Off-Path-SmartNICs handelt es sich hingegen um Geräte, bei denen eine klarere Trennung der Hardwareeinheiten vorgenommen wird. Eben diese Trennung ermöglicht eine Spezialisierung, mit der für bestimmte Anwendungszwecke besondere Hardwarebeschleuniger

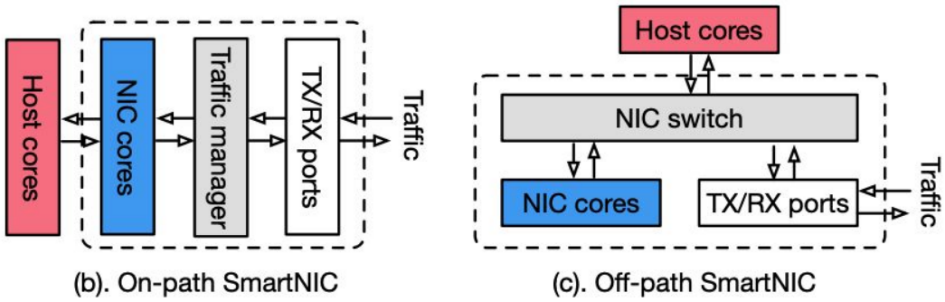

Abbildung 2.2: On-Path vs. Off-Path-Switching [28]

Abbildung 2.3: Firewire ASIC - VIA CT6306 [29]

verbaut werden können. Diese Architektur erlaubt es, einen kleineren Anwendungsbereich abzudecken, dieser wird aber sehr schnell verarbeitet. Wenn ein Paket eintrifft, muss es nicht von einem Prozessorkern der Netzwerkkarte beachtet werden, sondern kann von der speziellen Hardwareeinheit verarbeitet werden. [10] Dabei werden in aller Regel **Application Specific Integrated Circuits**, kurz ASICs, verwendet. Die Konfiguration dieser ASICs erfolgt dann meistens mittels Software, die auf dem Prozessor der Netzwerkkarte läuft. Allerdings gibt es dort auch Unterscheidungen, ob die Anwendung zur Konfiguration auf dem Hostsystem, also dem Rechner in dem die SmartNIC eingebaut ist, oder der Karte selbst ausgeführt werden soll. Off-path-SmartNICs verbessern somit die gesamte Architektur, indem sie die Allgemeinen Kerne des Prozessors klar vom kritischen Pfad der Netzwerkpakete trennen und es so verhindern, dass die logische Last, welche mit einer Anwendung einhergehen kann, sich weniger auf den tatsächlichen Durchsatz der Netzwerkkarte auswirken kann.

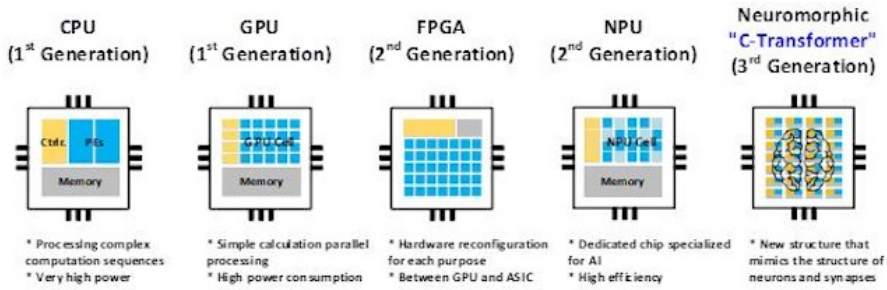

Abbildung 2.4: KAIST LLM Accelerator [27]

2.2.3 Application Specific Integrated Circuits

Application Specific Integrated Circuits (ASICs) sind Hardwareeinheiten, die ein sehr großes Anwendungsgebiet entwickelt haben. So werden sie nicht nur in Netzwerkbeschleunigern verwendet, sondern werden immer dann verwendet, wenn der Zweck der Hardware sich gut beschreiben lässt und somit keine große Varianz an Aufgaben von der entsprechenden Einheit übernommen werden muss. So wird es nämlich den entsprechenden Ingenieuren erlaubt, statt eines allgemeinen Rechners eine hochspezialisierte Einheit zu entwickeln, bei der die Merkmale einer klassischen Prozessorrecheneinheit vernachlässigt werden können. Dadurch ist eine wesentliche Effizienzsteigerung möglich, die es erlaubt, mit wesentlich weniger Energie die gleiche Leistung zu erzielen, wie sonst eine vergleichbare Recheneinheit [11]. Für rechenintensive Systeme hingegen lässt sich die gesteigerte Effizienz nutzen. Somit kann der allgemeine Durchsatz angehoben werden und es sind Systeme realisierbar, die mit der Anwendung klassischer Rechenwerke erheblich mehr Aufwand bedeuten würden. Zudem erzeugen solche Systeme deutlich mehr Abwärme und würden die Entwicklung somit auch vor ein thermodynamisches Problem stellen. Der Überbegriff ASIC ist nur sehr allgemein und gibt ähnlich wie bei Prozessoren keine Auskunft über die genaue Implementierung. Es gibt viele Architekturen, die für ASICs zur Anwendung kommen, da eben auch die Gebiete weitreichend sind und sich sehr divers unterscheiden. In Abbildung 2.3 ist ein ASIC zu sehen, der die Kommunikation zwischen einem Prozessor und einem sogenannten FireWire-Anschluss eines Endgerätes realisiert. Es muss festgelegt sein, welcher Standard von Geräten mit besagtem Anschluss erwartet wird, um den Prozessor des Hostgerätes mittels eines ASICs zu entlasten.

Der Zeitgeist in der Computerwissenschaft färbte letztlich außerdem das immens steigende Interesse an Machine Learning ab. Proportional dazu stieg das Interesse an effizienterer Hardware an, da etwaige Lernprozesse extrem energie- und somit kostenaufwendig sind. Es zeigt sich aktuell ein gesteigertes Interesse daran, ASICs zu fertigen, die effiziente hochdimensionale Vektoroperationen umsetzen. Dazu gibt es bereits mehrere Ansätze, mit denen beispielsweise Large Language Models auf einer ASIC-Architektur ausgeführt werden können (siehe Abbildung 2.4). Machine Learning bietet großes Potenzial, Energie mittels ASICs zu sparen, da bisher GPUs und CPUs zum Einsatz kamen, die beide große Energiekosten mit sich bringen. So erlauben ASICs eine extreme Effizienz im Hinblick auf

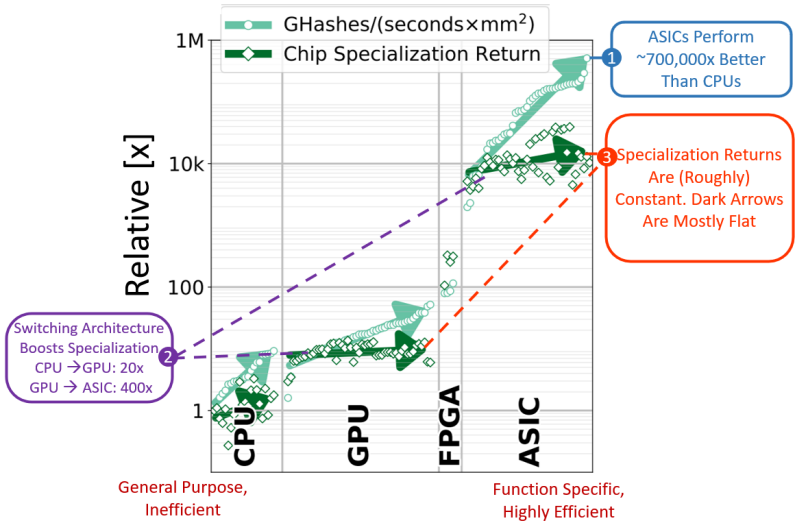

Abbildung 2.5: Hardware-Architekturen beim Bitcoin Mininig [26]

Leistung/Energiekosten.

Abbildung 2.5 zeigt die Leistungsniveaus der unterschiedlichen Architekturen bei einem Bitcoin-Mining-Algorithmus. Im Wesentlichen wird beim Mining etwaiger Kryptowährungen eine Hashsumme berechnet, mit der ein Teil der Blockchain validiert wird. Dabei wird im Falle von Bitcoin ein SHA-256 Hash verwendet. Da es sich dabei um eine sehr allgemeine Berechnung handelt, kann ein guter Vergleich zu vielen anderen Berechnungen gezogen werden. So zeigt Abbildung 2.5, wie sehr in manchen Fällen Algorithmen hinsichtlich ihrer Laufzeit von speziellen Architekturen profitieren können.

2.2.4 Speedup

Im Allgemeinen werden die Leistungen in eine Metrik gegossen, damit ein objektiver Vergleich vorgenommen werden kann. Dazu entwickelte sich der allgemeine Speedup, der beschrieben wird durch: [12]

$$Speedup = \frac{T_{alt}}{T_{neu}} \tag{2.1}$$

Dabei ist:

- T_{alt} die Ausführungszeit vor der Optimierung bzw. der Vergleichshardware
- T_{neu} die Ausführungszeit nach der Optimierung bzw. der Vergleichshardware

Um einen Vergleich zwischen den unterschiedlichen Lastverteilungsarchitekturen aufzuzeigen, wird im Folgenden der allgemeine Speedup verwendet.

3 Related Work

Um diese Arbeit in einen wissenschaftlichen Kontext zu setzen, wird im Folgenden ein Überblick über aktuelle Arbeiten aus dem Feld SmartNIC und BlueField gegeben.

3.1 Demystifying Datapath Accelerator Enhanced Off-path SmartNIC

Diese Arbeit wurde 2024 von Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu und Zeke Wang verfasst. Sie ist an der Zhejiang University, Alibaba Cloud und der University of Wisconsin–Madison entstanden.[31]

In diesem Paper wird ein allgemeiner Performance-Kontext erstellt. Dabei liegt der besondere Fokus auf der Analyse und der Leistung des Datapath Accelerators (DPA). Dieser ist auf allen Generationen BlueField verbaut worden, wird allerdings kaum in dem Material von NVidia erwähnt. Allerdings handelt es sich beim DPA um die relevanteste Hardwareeinheit für die Netzwerkfunktionalität.

In der Arbeit wird auch ein Vergleich zwischen dem Host-System, dem ARM-System und dem DPA selbst gezogen. Dabei werden die Metriken Latenz, verzerrte Lasten, Schreib- und Lesebandbreite, Sende- und Empfangspaketbandbreite u. a. ausgewertet (siehe Abbildung 3.1). Die Autoren identifizieren den verbauten DPA als einen RV64IMAC. Dies ist ein RISC-V-Prozessor mit einem Basistakt von 1.8 GHz. Sie entwickeln drei unterschiedliche Benchmark-Methoden, um eine Baseline zu schaffen, die alle drei verfügbaren Architekturen sinnvoll miteinander vergleicht. Die Autoren erkennen den DPA als einen der eventuellen Schwachpunkte einer spezifischen Implementierung. Die besondere Kritik

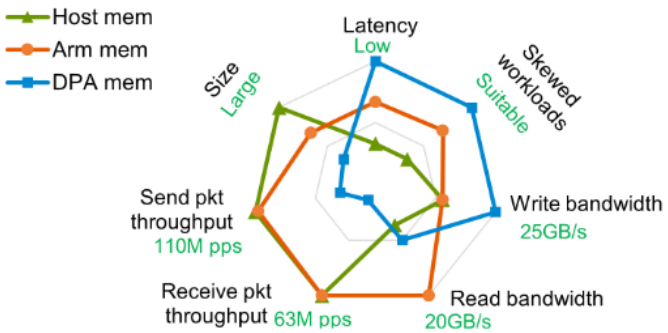

Abbildung 3.1: Vergleich der Speicher der Architekturen [31]

liegt dabei auf der fehlenden Dokumentation seitens NVidia, die es so den Entwickelnden der Plattform deutlich erschwert, Software für diese zu entwickeln.

3.2 Performance Characteristics of the BlueField-2 SmartNIC

Diese Arbeit wurde 2021 von Jianshen Liu, Carlos Maltzahn, Craig Ulmer und Matthew Leon Curry verfasst. Sie ist an der UC Santa Cruz und den Sandia National Laboratories entstanden.[33]

Abermals geht es in dieser Arbeit um die allgemeine Performance der BlueField-2. Allerdings wird hier neben der gegebenen Hardware auch die Netzwerkfähigkeit überprüft. Die ARM-Kerne werden mittels Netzwerkaufgaben stark belastet, um die oberen Grenzen für Abhängigkeiten im Datenfluss zu erkennen. Das Paper ist von besonderem Interesse im Kontext dieser Bachelorarbeit, da die echte Leistung der BlueField-2-Generation mit der beworbenen Netzwerkbandbreite verglichen wird. Die Autoren implementieren einen Testfall, bei dem Netzwerkverkehr zwischen dem Host, der Smart NIC und einem weiteren Remote-Host stattfindet. Sie kommen zu dem Ergebnis, dass nur sehr schwer und unter einer Menge von Umständen überhaupt die volle Netzwerkbandbreite verwendet werden kann. Hauptsächlich ist die Leistung von der spezifischen Arbeitslast abhängig. Somit sollte vor einer Implementierung abgewogen werden, um festzustellen, ob sich das Offloading überhaupt auf eine Smart NIC-Architektur lohnt.

3.3 A Comprehensive Survey on SmartNICs: Architectures, Development Models, Applications, and Research Directions

Diese Arbeit wurde 2024 von Elie Kfoury, Samia Choueiri, Ali Mazloum, Ali AlSabeh, Jose Gomez und Jorge Crichigno verfasst. Sie ist an der University of South Carolina entstanden.[34]

Dieses Paper stellt einen guten Überblick über Smart NICs im Allgemeinen dar. Außerdem wird die Entwicklung der Architekturen der SmartNICs historisch aufgearbeitet. Es kann als eine Art Metapaper betrachtet werden, da auch ein Ausblick gegeben wird, was in Zukunft auf den jeweiligen Plattformen von Interesse sein könnte. Es wird ein klarer Zusammenhang zwischen dem Ende des Mooreschen Gesetzes und der fehlenden weiteren Leistungssteigerungen bei normalen Prozessoren mit dem Aufkommen von Smart NICs hergestellt. Es wird zwischen den drei folgenden Typen von Netzwerkkarten unterschieden:

- Network Interface Cards beschreiben klassische Netzwerkkarten ohne weitere programmierbare Funktionalität.
- Offload-NICs bieten eine Einbettung von einigen Hardwarebeschleunigern für bestimmte Aufgabenbereiche.
- Smart-NICs beschreiben die Kombination aus klassischen Prozessoren mit anwendungsbezogenen speziellen Prozessoren und Hardwarebeschleunigern.

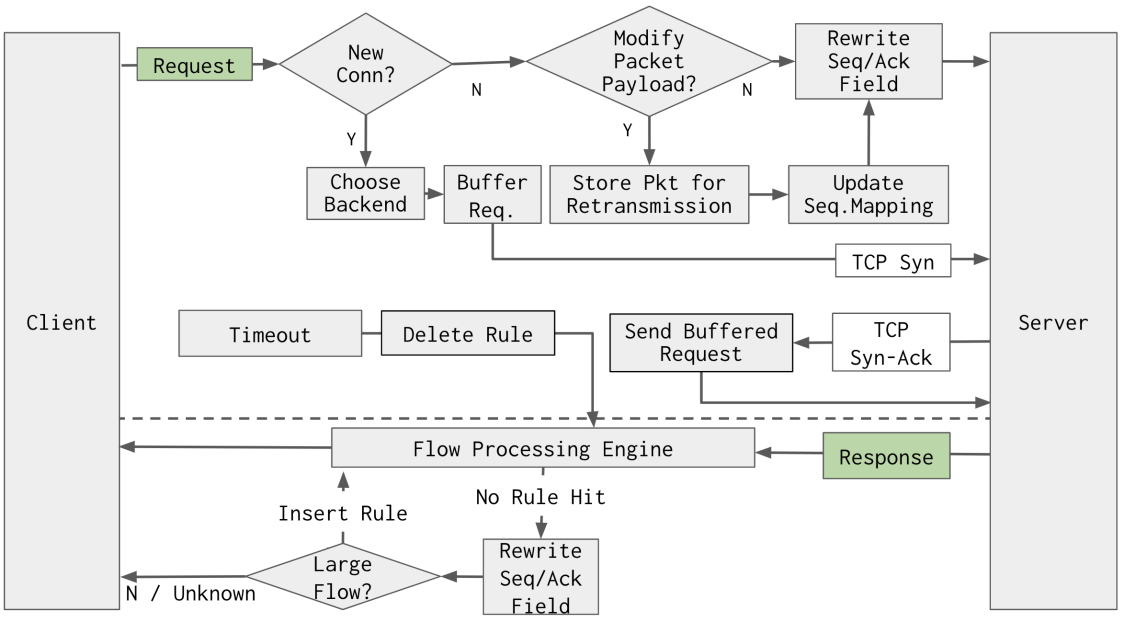

Abbildung 3.2: Leichtgewichtiger Netzwerkstack des Laconic LB

Innerhalb der Klasse von Smart-NICs treffen die Autoren außerdem die weitere Unterscheidung zwischen FPGA- und ASIC-basierten Smart NICs. Es wird angeraten, eine standardisierte Programmierungsplattform zu entwickeln, die es erlaubt, für alle Smart NIC-Plattformen zu programmieren.

3.4 Laconic: Streamlined Load Balancers for SmartNICs

Diese Arbeit wurde 2024 von Tianyi Cui, Chenxingyu Zhao, Wei Zhang und Kaiyuan Zhang verfasst. Sie ist an der University of Washington und bei Microsoft entstanden.[35]

Dieses Paper kommt dieser Arbeit auf fachlicher Ebene wohl am nächsten und wäre von Seiten des Quelltextes eine gute Referenz gewesen. Leider wurde der besagte Quelltext allerdings nicht veröffentlicht. Somit konnte kein Bezug auf diese Arbeit genommen werden. In ihrer Arbeit beschreiben die Autoren, wie wichtig L7-Lastverteilung für moderne Workloads geworden ist. Dazu stellen sie einen Bezug zu den Energiekosten von aktuellen L7-Lastverteilern her, wobei natürlich die hohen Energiekosten im Mittelpunkt stehen. Die meisten Optimierungen von Laconic mittels Hardwarebeschleuniger wurden laut den Autoren vor allem auf L4 des OSI-Modells durchgeführt. Deshalb ist ihr Ansatz, auf einer Smart-NIC-Architektur einen besagten Lastverteiler zu implementieren. Als Referenz wählen sie dazu den Nginx als Softwarelastverteiler. Der Lastverteiler wird von den Autoren Laconic genannt. Dieser beinhaltet auch einen komplett eigenen Netzwerkstack auf der BlueField, der so laut den Autoren komplett ohne TCP/IP-Stack arbeiten kann (siehe Abbildung 3.2). Besagter Stack erlaubt es ihnen, sehr schnell und vor allem hardwarebeschleunigt auf der Smart NIC zu verteilen. Dazu setzen sie stark auf gute Parallelität der Algorithmen, um möglichst effizient die zur Verfügung stehenden 256 Threads des DPA zu

verwenden. Sie zeigten also erstmalig, dass ein leichterer Stack, der speziell für die entsprechende Hardware entwickelt wurde, wesentlich schneller sein kann als eine vergleichbare Anwendung auf einem x86-Prozessorkern. Sie erreichen einen Speedup von 8.7 gegenüber dem klassischen Nginx.

4 NVIDIA Bluefield-3

Die im Rahmen dieser Arbeit verwendete SmartNIC ist die von Nvidia im Jahre 2022 veröffentlichte Bluefield-3. Wie bereits am Namen der Karte zu erkennen ist, handelt es sich dabei um die dritte Generation von Netzwerkkarten, die zusätzlich zur netzwerkspezifischen Architektur noch einen allgemeinen ARM-Prozessor verbaut hat. Zusätzlich wird in einem späteren Abschnitt ein Überblick über das DOCA-Framework gegeben, welches von NVidia als einheitliche Programmierschnittstelle veröffentlicht wurde, um die Programmierung der Hardware von der jeweiligen Generation unabhängig zu machen.

4.1 Entstehung

Die Entwicklung der BlueField ist ein Produkt der letzten Dekade, in der ein immerwährend größeres Interesse an der Effizienzsteigerung von Rechenzentren entstanden ist. Energiesparende Cluster, die dieselbe Leistung erreichen wie selbige Integration auf einem allgemeinen Prozessor, sind somit für alle Beteiligten eines Rechenzentrums interessant.

4.1.1 BlueField-1

Die erste Generation der Bluefield-Hardware (siehe Abbildung 4.1) wurde im Jahre 2017 von Mellanox Technologies vorgestellt und bildet den ersten Eintrag in die Reihe der Bluefield-SmartNIC-Serie. Sie kombinierte erstmalig die ConnectX-5-Plattform mit einem ARM-basierten System-on-Chip. Die Idee dahinter war es, das Verhalten der Netzwerkplattform mittels User-Anwendungen beeinflussen zu können. Dazu wurde eine API zwischen der ConnectX und dem ARM-Prozessor entwickelt. Hierzu sei erwähnt, dass es sich bei der ConnectX-Plattform um eine bereits weit verbreitete und in viele Systeme integrierte Hardware handelte. Sie besaß bereits zum damaligen Zeitpunkt eine breite Anzahl von paketbasierten Operationen, mit deren Hilfe eine weitreichende Manipulation sowie Paket-Steering oder anderweitige Verwendungen ermöglicht wurden. Die BlueField-1 war mit einem ARM-A72 mit 4 Kernen und 8 Threads ausgestattet und besaß außerdem zwei QSFP28-Anschlüsse, welche bis zu 100 Gbps Durchsatz erreichen konnten. Die Zielgruppe der BlueField-Serie waren von Anfang an der Enterprise-Markt sowie die zahlreichen größeren Rechenzentren, in denen Rechenoperationen ausgeführt werden, die auf mehreren Computern eines Clusters oder einer Clusterstruktur ausgeführt werden. Gerade in diesem Umfeld wird eine derartige Architektur von Interesse, da der Netzwerkverkehr so anwendungsspezifischer verteilt werden und somit die Effizienz eines Rechnerverbundes steigern kann. [13]

Abbildung 4.1: BlueField-1-Karte [30]

4.1.2 BlueField-2

Nach der Übernahme von Mellanox durch den Chip-Hersteller NVidia im Jahre 2019 wurde das Projekt BlueField von NVidia nicht nur weitergeführt, sondern mit dem Aufkommen von vermehrten Machine Learning Workloads sogar das Marketing ausgebaut. So wurde die BlueField-2 im Jahre 2021 unter dem neuen Hersteller NVidia veröffentlicht und bot nun einen QSFP56, der laut eigenem Marketing-Material bis zu 200 Gbps erreichen sollte. Abermals kam der ARM-A72 zum Einsatz, der erneut mit 4 Kernen und 8 Threads verbaut wurde. [13] Erstmalig wurde außerdem damit geworben, dass ein Einsatz in Machine-Learning-Rechenclustern besonders wertvoll sei. Auf Angaben, warum genau das der Fall sein sollte, wurde seitens des Herstellers verzichtet. Insbesondere nicht vor dem Hintergrund, warum es sich dabei nun um ein Alleinstellungsmerkmal anderen SmartNICs gegenüber handeln sollte.

4.1.3 BlueField-3

Zuletzt wurde im Jahre 2022 die BlueField-3 veröffentlicht, die im Rahmen dieser Arbeit zur Verwendung kam. Erstmalig wurde nun der ARM-A78 mit 8 Kernen und somit 16 Threads verbaut. Die Arbeitsspeicherarchitektur nahm zusätzlich auch den Generationensprung von DDR4 auf DDR5 vor und verwendet so einen deutlich schnelleren Speichertakt als die vorherige Generation. Außerdem wurde erneut der Netzwerkanschluss aktualisiert und verwendet nun den QSFP112. Laut NVIDIA soll mit diesem Netzwerkverbund eine

Line Rate von bis zu 400 Gbps erreicht werden. [14] Dabei wird im Marketing-Material nicht explizit erwähnt, welche Paketgröße für besagten Durchsatz verwendet wurde. Erneut soll laut NVIDIA der Fokus der Hardware vermehrt auf dem Einsatz in Rechenzentren liegen, in denen größere Machine-Learning-Lasten laufen. Die BlueField-3 soll daraufhin die Rolle einer intelligenteren Paketverteilung einnehmen, wobei die Last der Lastverteilung nicht mehr auf dem Hostsystem bzw. dem entsprechenden Host-Prozessor liegt, sondern eben auf der Netzwerkkarte selbst. Außerdem sind eine Menge von weiteren speziellen hardwarebeschleunigten Hardwareeinheiten auf der neuesten Iteration der BlueField-Serie verbaut worden. Alle genannten Generationen BlueField werden per PCI-E-Stecker in den Hostsystemen verbaut und verwenden so den aktuellsten PCI-E 5.0 Standard, der auf eine theoretische Maximalbandbreite von 32 Giga Transfer/s kommt. Somit soll eine ultraschnelle Schnittstelle zwischen Hostsystem und BlueField-3 erreicht werden. Dazu ist in Abbildung 4.2 der Datenpfad abgetragen, an dem zu erkennen ist, dass nachdem ein Paket den gesamten Weg durch die BlueField genommen hat, schlussendlich auf dem Host-System (hier grün) ankommen kann.

4.2 Architektur

Um der Funktion einer intelligenten Netzwerkkarte nachzukommen, sind auf der BlueField diverse Hardwareeinheiten verbaut, die eine Reihe von Einsatzzwecken abdecken sollen. Dabei werden nicht alle angebotenen Funktionen auch tatsächlich von ASIC-Hardwareeinheiten übernommen, sondern werden teilweise komplett oder nur in bestimmten Pipelineabschnitten auf dem ARM-Core ausgelagert, um dort einer weiteren Behandlung unterzogen zu werden. Wie in Abbildung 4.2 zu erkennen ist, gliedert NVIDIA die BlueField-Karte grob in Domänen unterschiedlichster Funktion auf. Abgesehen vom ARM-Teil der Karte, womit im Wesentlichen der Bereich beschrieben wird, in dem der Prozessor, Arbeitsspeicher und restlicher I/O zusammengefasst werden, sind die beiden Bereiche **Accelerators** und **Accelerated Programmable Pipeline** diejenigen, in denen eine große Menge der eigentlichen Hardwarebeschleuniger sitzen. Allerdings geht aus dem von NVIDIA bereitgestellten Material nicht vollständig hervor, worum es sich bei dem sogenannten Datapath Accelerator handelt. [15] Im Release-Flyer der BlueField-3 steht allerdings, dass es sich dabei auch um einen Prozessor mit 16 Kernen handeln soll, der mit 256 Threads ausgestattet ist. Dies würde bedeuten, dass jeder Kern von Hause aus 16 Threads besitzt. Zusätzlich wird oft darauf verwiesen, wie einfach ein Zusammenspiel von DPU und Hostsystem mittels der PCI-E-Anbindung funktionieren soll. Im Folgenden wird eine Auswahl aus Beschleunigern und Funktionen vorgestellt, die vom Hersteller angeboten werden. [15]

4.2.1 Hardwareeinheiten

Im Folgenden wird eine Auswahl von Hardwarebeschleunigern vorgestellt.

PKA

PKA steht für Public-Key-Acceleration und bietet eine Reihe von Funktionalitäten im Zusammenhang mit der Arbeit mit öffentlichen Schlüsseln im Kontext von SSL. So stellt es

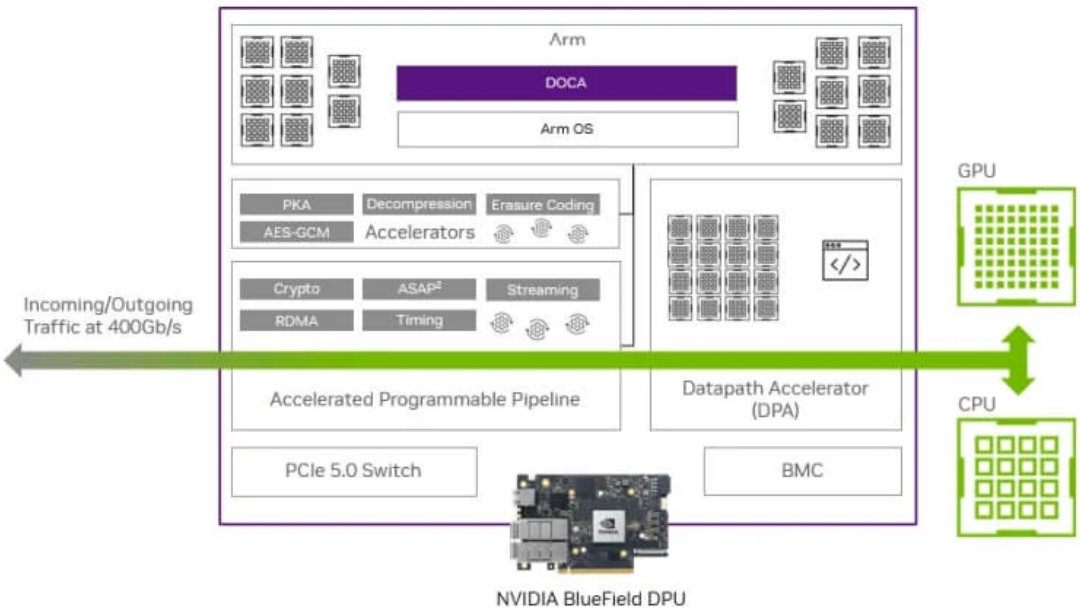

Abbildung 4.2: BlueField-3 Architektur [36]

beispielsweise einfache bis komplexere arithmetische Operationen wie RSA, Diffie-Hellman sowie Addition und Subtraktion bereit. Es soll eine einfache Programmierschnittstelle bieten, sobald die DOCA-Applikation mit Verschlüsselungen hantiert, die möglichst schnell umgesetzt werden sollen.

Decompression

Die Dekompressionsbeschleuniger der BlueField-3 sollen vor allem dann zum Tragen kommen, wenn die Netzwerkkarte mit Buffern konfrontiert wird, in denen die Daten komprimiert vorliegen. Dabei soll sich diese möglichst nicht auf die Leistung der Verarbeitung des Datenverkehrs auswirken.

Erasure Coding

Erasure Coding bezeichnet die Hardwareeinheit, deren hauptsächliche Funktion die Forward Error Correction ist. Dies wird dazu verwendet, um eventuelle Übertragungsfehler oder umgedrehte Bits mithilfe der Paritätsinformation reparieren zu können.

AES-GCM

Advanced-Encryption-Standard (AES) ist ein bekannter Verschlüsselungsstandard, der quasi von jedem Endgerät dieser Welt implementiert wird, um modernen Sicherheitsanforderungen gerecht zu werden. Hierzu hat NVidia eine Hardwareeinheit entwickelt, um Daten

auch im eigenen Arbeitsspeicher verschlüsseln zu können.

RDMA

RDMA ist die Abkürzung für Remote Direct Memory Access. Innerhalb eines solchen Prozesses kann ein Prozess auf den Speicherbereich eines anderen Rechners zugreifen. Damit wird eine Alternative zum klassischen Netzwerkverkehr angeboten.

4.2.2 Datapath Accelerator

Der Datapath Accelerator, kurz DPA, ist für diese Arbeit von besonderem Interesse, da laut NVidia hier ein Großteil der Netzwerkverarbeitung vorgenommen wird. Wie bereits im vorherigen Kapitel zum Hardwareaufbau erwähnt, soll hier ein weiterer Prozessor zum Einsatz kommen, der über ein extremes 16-faches Multithreading verfügt. Dieses Multithreading wird benötigt, um möglichst latenzfreie Verarbeitung umzusetzen. [16] Leider sind seitens des Herstellers allerdings keine weiteren Angaben zur verbauten Hardware freigegeben worden. Daher ist es ohne weitere Informationen nicht möglich, die genauere Funktionsweise zu analysieren. Dennoch sollte es, sofern es sich dabei denn wirklich um den trafficverarbeitenden Prozessor handelt, in den späteren Messungen zu beobachten sein, dass der ARM-Prozessor von dem eingehenden Verkehr unberührt bleiben sollte.

4.3 DOCA Framework

DOCA ist ein von NVidia entwickeltes Framework, das speziell für die BlueField-Karten entwickelt wurde. Es gliedert sich in verschiedene Teilbereiche und unterscheidet dabei zwischen architektonischen Grundsätzen, Programmierinterfaces, aber auch dem Betriebssystem für die Karte selbst. Allgemein ist DOCA dafür gedacht, der direkte Anlaufpunkt für Entwickler zu sein, die auf der BlueField-Plattform entwickeln wollen. Damit werden auch Dinge wie Programmiersprachen, Programmierparadigmen und eine Reihe von bereits angelegten Bibliotheken bereitgestellt. Zusätzlich werden sämtliche Hardware-Treiber ebenfalls unter dem Oberbegriff DOCA zusammengefasst. [17] Im Folgenden wird ein Überblick zu DOCA im Allgemeinen gegeben und ein besonderes Augenmerk auf die für diese Arbeit relevanten Teilbereiche gelegt. In Abbildung 4.3 ist ein Überblick aus der NVidia-Dokumentation zu sehen, in der die Unterscheidung in Services, Bibliotheken und Treibern hervorgehoben ist.

4.3.1 DOCA Services

DOCA stellt eine Reihe von Services bereit, die vorrangig dazu dienen sollen, eine Ende-zu-Ende-Lösung für einen speziellen Anwendungsfall zu bieten. Dazu gliedert NVidia diverse (Abbildung 4.3) Unterkategorien aus. Aufgrund des primären Anwendungsfeldes in Rechenzentren wird viel Wert auf Orchestrierung gelegt. Leider bleibt NVidia hier bemerkenswert ungenau, was genau damit eigentlich gemeint ist. Bei der Telemetrie hingegen stellt NVidia eine große Schnittstelle bereit, um genaue hardwarebezogene Daten abrufen zu können. Hierzu werden den Entwickelnden zwei Tools bereitgestellt. Das Tool **do-**

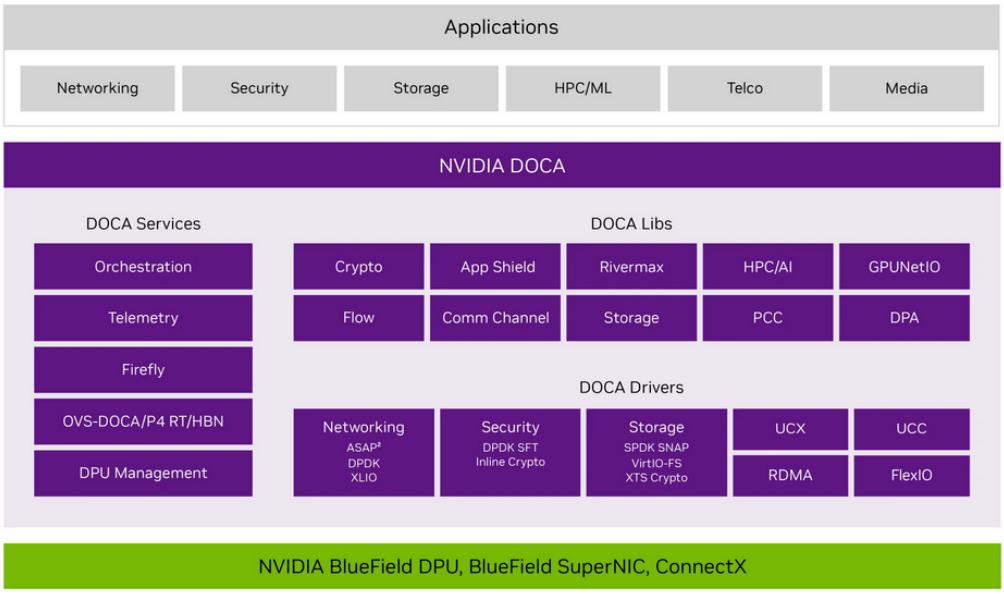

Abbildung 4.3: Überblick der Teilbereich von DOCA [17]

ca_telemetry_diag stellt die Konfiguration des Loggings dar. Dazu kann mithilfe besagter Tools festgelegt werden, welche Telemetriedaten und auch in welchem Samplingintervall erhoben werden sollen. Diese Daten können dann in diversen auswählbaren Dateiformaten abgerufen werden. **doca_telemtery_pcc** soll den Zugang zu speziellen algorithmischen Daten ermöglichen. Damit sind die bereits in DOCA vorimplementierten Funktionen gemeint, die meist hardwarebeschleunigt, also außerhalb des sichtbaren Bereichs des ARM-Hosts, ausgeführt werden. Sonstige Services wie beispielsweise der Firefly sind NVidias Implementierungen des Precision Time Protocols. Ziel dieses Protokolls ist es, die Zeit innerhalb eines Clusters möglichst genau propagieren zu können. Dazu verspricht NVidia, möglichst viel von dieser Komplexität vor dem Anwender mithilfe von Firefly fernzuhalten. Außerdem setzt NVidia zur Konfiguration des Datenverkehrs stark auf die Integration von Open vSwitch (OVS). Dazu wurde eigens ein eigener Service entwickelt, der abermals die eigentliche Komplexität des Konfigurationsprozesses dem Entwickler abgenommen werden soll. Hauptgrund ist allerdings vermutlich, dass, um mit den spezifischen Hardwareeinheiten überhaupt kommunizieren zu können, die Open vSwitch-Interfaces angepasst werden müssen.

4.3.2 DOCA Bibliotheken

Damit eine reibungslose Inbetriebnahme sowie Programmierung einer SmartNIC erfolgen kann, muss klar definiert sein, wie, sofern programmatische Konfigurationen erfolgen sollen, ein Entwickler mit der entsprechenden Hardwareeinheit kommunizieren kann. Damit ist es nicht mehr nötig, die Feinheiten der einzelnen Hardwarekommunikationskanäle zu

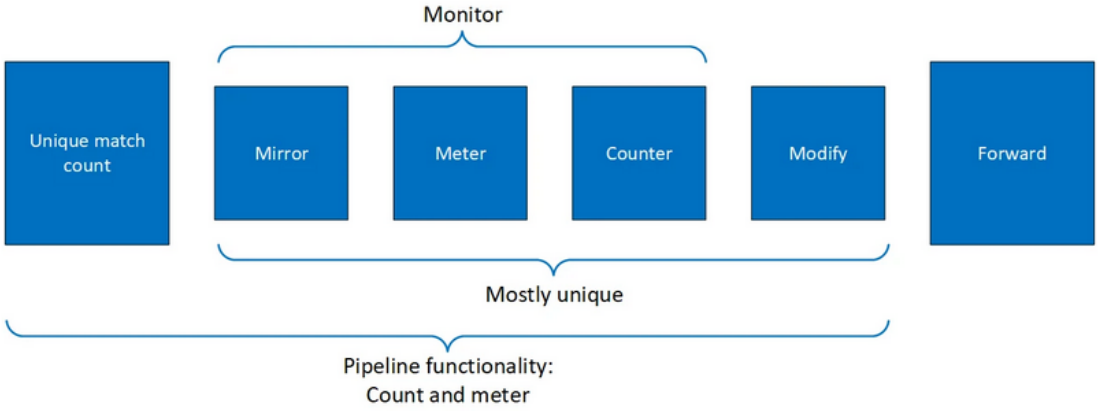

Abbildung 4.4: Schemata und Zugehörigkeiten einer Flow Pipe [18]

implementieren, sondern es kann auf eine mehr oder weniger vereinheitlichte Programmierschnittstelle zugegriffen werden. Dazu stellt NVidia Programmbibliotheken in Form von C-Headern bereit, mithilfe derer intrinsische Funktionen auf der Architektur ausgeführt werden können. Dabei lässt sich grob formulieren, dass jede in Kapitel 4.2.1 genannte Hardwareeinheit eben auch eine entsprechende Bibliothek bekommen hat. Zusätzlich gibt es aber auch Bibliotheken, die Funktionen wie Speicherverarbeitung für Anwendungsfälle wie verteilte Speichersysteme implementieren.

4.3.3 DOCA Flow

DOCA Flow ist die Bibliothek, die die Verarbeitung und Modifikation von Netzwerkpaketen enthält. Das Versprechen seitens NVidia ist hierbei, dass alle Funktionen von Flow auf den Hardwarebeschleunigern ausgeführt werden können. DOCA Flow wird, wie viele moderne Systeme, deklarativ programmiert. Das bedeutet, der State des aktuellen Netzwerkverkehrs ist immer genau so, wie er durch die Konfiguration deklariert wurde. Dies hat den großen Vorteil, dass die Entwicklung und in der Folge auch das Debugging sich immer direkt mit dem Programmcode selbst befassen kann, da keine Fehler zur Laufzeit entstehen können (sollten) die so nicht klar vorher definiert worden sind. Allgemein ist DOCA Flow darauf ausgelegt, statt Pakete einzeln zu behandeln, in Netzwerkdatenpfaden den Verkehr zu steuern. Dazu werden die Hardwareregeln wie beispielsweise **Packet Matching**, **Forwarding** und **Monitoring** definiert. [18]

Flow Pipe

Eine Flow Pipe beschreibt einen genannten Datenpfad, der deklarativ an die BlueField-3 API übergeben wird. Dabei setzt sich eine Flow Pipe immer aus den vier wesentlichen Bestandteilen **Match**, **Monitor**, **Action** und **Forward** zusammen (siehe Abbildung 4.4). Diese Bestandteile können in **Entries** organisiert werden. Sie bilden die tatsächlichen aktiven Teile der Pipe. Pipes bilden eine Abstraktion eines Datenpfades, auf denen die einzelnen Entries angewendet werden.

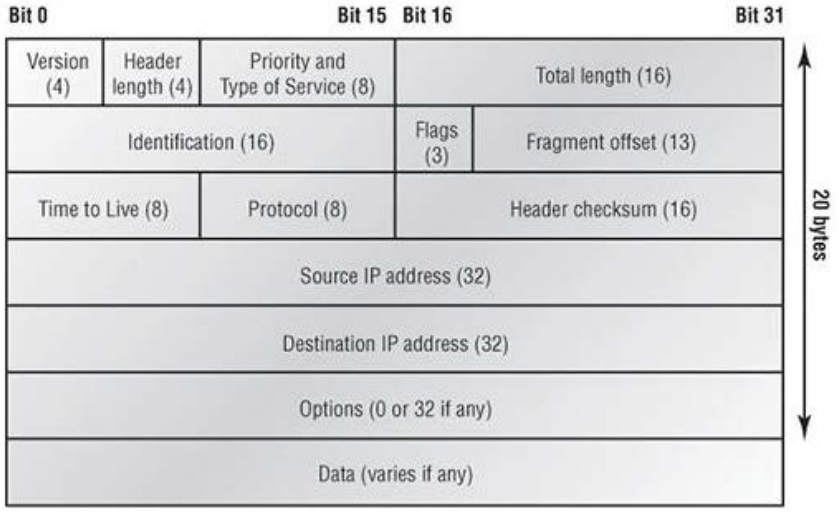

Abbildung 4.5: Aufbau eines IP-Headers [37]

Beim Match wird mithilfe einer Zusammensetzung aus Filtern und Match-Regeln klar definiert, welche Pakete aus dem Datenstrom der BlueField genommen werden sollen und welcher entsprechenden Pipe zugeordnet werden.

Daraufhin wird das Paket laut Dokumentation gespiegelt und dem Monitorteil übergeben. Dieser erlaubt es, Daten über die Pipe auszulesen. So kann zur Laufzeit die Anzahl der von der Pipe verarbeiteten Pakete sowie deren Größe in Bytes ausgelesen werden. Im Verlauf der Messung im späteren Teil wurde der Versuch durchgeführt, ob ein ein- bzw. ausgeschaltetes Monitoring Auswirkungen auf die Leistungsfähigkeit der Karte hat. Dies ist klar zu verneinen. Weder Bandbreite noch Pakete/Sekunde wurden von dem Monitoring beeinflusst.

Nachdem nun Pakete in der entsprechenden Flow Pipe ankommen, können nun eine Reihe von Aktionen auf diese ausgeführt werden. Diese Funktionen werden in Entries durchgeführt. Darunter befinden sich vor allem Funktionen, die die Ethernet- sowie IP-Header modifizieren können. Es ist so beispielsweise nicht ohne Weiteres möglich, Pakete einzeln zu verarbeiten, sondern nur Klassen von Paketen, die eben in der entsprechenden Pipe bzw. dem entsprechenden Entry angekommen sind. Die eigentliche Payload des L7-Paketes kann **nicht** von Flow verarbeitet werden. Dies ist vor allem der Tatsache geschuldet, dass es sich hierbei eben um die Programmierung von Hardwarebeschleunigern handelt. Würden wir Pakete einzeln verarbeiten wollen, so müssten diese zur Laufzeit auf den Prozessor der SmartNIC gebracht werden, was mit erheblichen Leistungseinbußen einhergehen würde. Die Payload der Pakete kann nicht verarbeitet werden, da diese eine variable Länge besitzen kann, was ein Problem für eine feste Hardwarearchitektur darstellt. Bei festgelegten Header-Bitfeldern allerdings kann eine festdefinierte ASIC-Hardware gut die entsprechenden Bits mithilfe von einfachen logischen Bauteilen verarbeiten [19]. In Abbildung 4.5 ist zu erkennen, dass sich bestimmte Teile eines Headers immer an genau der gleichen Stelle im

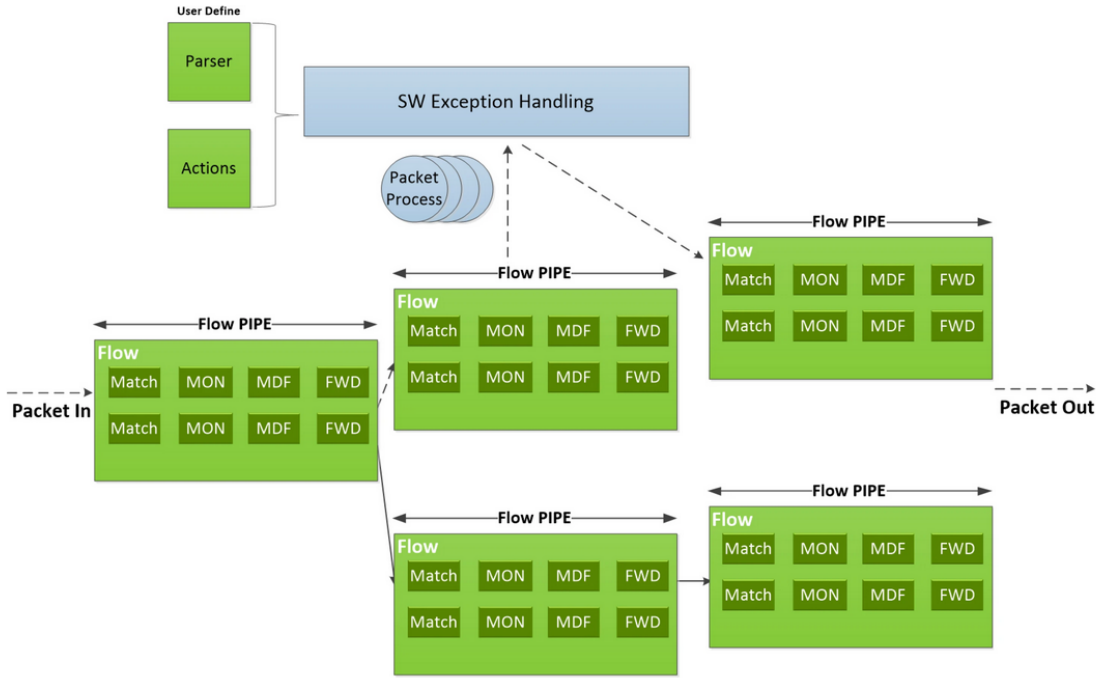

Abbildung 4.6: Beispielhafter Flow-Pipe-Aufbau [19]

Bitfeld befinden. So muss die Hardware nicht erst parsen bzw. erkennen, wo im Paket sich welche Daten befinden. Sie kann einfach davon ausgehen, dass bestimmte Informationen an bestimmten Stellen der Bitkette des Headers liegen.

Zuletzt kann in einer Pipe aber auch in den Entries selbst mittels des **Forward** festgelegt werden, wohin die relevanten Pakete geleitet werden sollen. Dabei kann sowohl direkt auf einen Port, der einem entsprechenden Hardware-Port auf der Karte gleicht, eine andere Pipe oder gar kein Folgeziel gewählt werden. Bei letzterem handelt es sich logisch um einen Drop des Pakets. Von besonderem Interesse ist allerdings die Funktion, in eine andere Pipe weiterzuleiten, da sich so komplexere Anwendungen umsetzen lassen, in denen einzelne Pipes bestimmten Traffic verarbeiten und mehrere Datenströme modifiziert werden können. Allgemein bietet DOCA Flow die Kategorisierung dieser Pipes. Dabei wird zwischen sogenannten **Steering Domains** unterschieden. Diese verbieten gewissen Pipefunktionen, wenn der entsprechenden Domain eine Pipe hinzugefügt wurde.

In Abbildung 4.6 ist eine Beispielanwendung per Diagramm zu sehen, in der eine Verkettung von Flow Pipes erfolgt ist. Es ist auch möglich, dass, wenn Pakete nicht von einer entsprechenden Pipe aus dem Datenfluss entnommen (also verarbeitet) werden, sie an einen Hostkern durchgereicht werden. Dort kann dann ein weiteres Programm laufen, welches daraufhin diese Pakete verarbeitet. Dabei verlässt das Paket allerdings den hardwarebeschleunigten Bereich und ist wieder an die Performance des ARM-Cores gebunden.

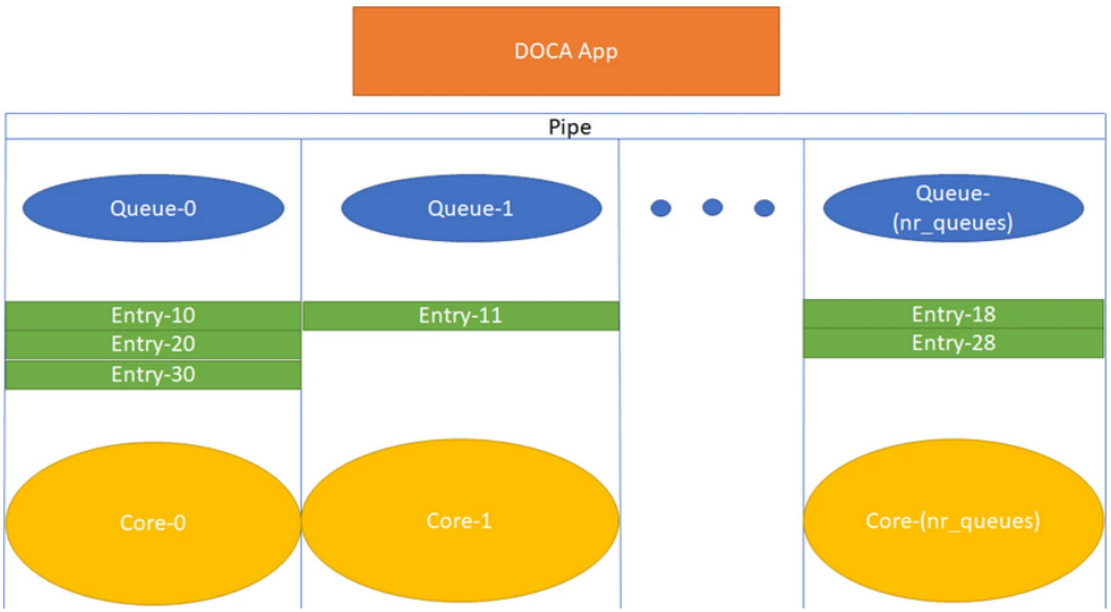

Abbildung 4.7: Auslagerung von Pipe Entries [19]

Flow Entry

Laut NVidia werden Entries einer Flow Pipe auf mehreren Kernen des Data Accelerators verarbeitet. Jedem dieser Kerne ist eine entsprechende Queue mit den zu verarbeitenden Paketen zugeordnet (siehe Abbildung 4.7). Durch diese Parallelisierung soll es möglich sein, dass der Paketverkehr ohne Latenzverzögerung verarbeitet werden kann. Leider waren im Rahmen dieser Arbeit keine weiteren Daten, wie genau diese Lastverteilung auf den 256 Threads funktioniert, auffindbar. Flow Entries können spezielle Aufgaben für eine Pipe übernehmen, sind allerdings lose an die Pipe-Definitionen gebunden. Soll beispielsweise in einem Entry entschieden werden, ob an Port 0 oder Port 1 weitergeleitet werden soll, so muss in der Pipe das entsprechende Forward-Feld auf 0xffff gesetzt sein, damit die Hardware weiß, dass der Forwardschritt erst im Entry erfolgt. Selbiges gilt natürlich für Matching und Actions.

Packet Matching

Für die Funktion eines Lastverteilers ist eine der Hauptfunktionen, den gesamten Traffic in handhabbarere Teillasten zu modulieren, damit diese dann auf verschiedene Backends verteilt werden können. Dazu ist eine Art des Packet Matchings besonders praktisch. Das Packet-Matching aus DOCA Flow ist in Abbildung 4.8 zu sehen. Zunächst erreicht ein Paket den Filter und wird mit diesem auf der Bit-Ebene verundet. Sprich, das Ausgabebitfeld besitzt an einer Position eine 1, wenn das Datenpaket und der Filter an der gegebenen Stelle auch eine 1 gesetzt haben. So kann eine Art Vorauswahl realisiert werden, in der Entwickler festlegen können, welche Bits des entsprechenden Feldes überhaupt analysiert werden sollen. Daraufhin erreicht das neu berechnete Bitfeld die Regel. In dieser wird über-

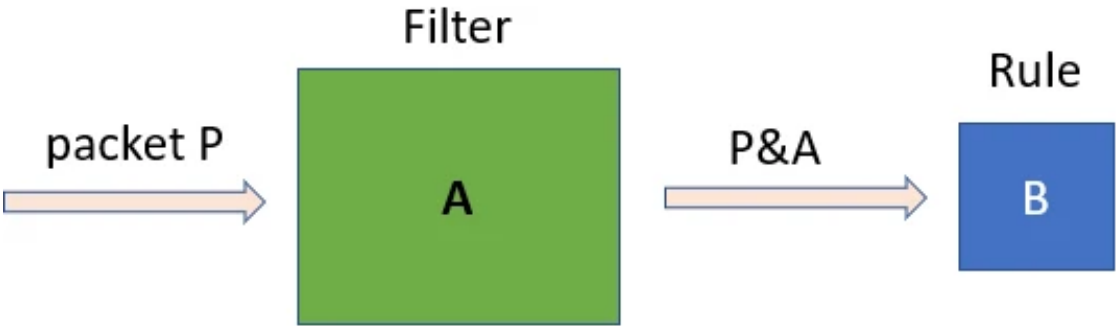

Abbildung 4.8: Matching Schema der BlueField [38]

prüft, ob das Bitfeld der Regel entspricht. Wenn dies der Fall ist, so wird das Paket dem Match zugeordnet. Wird die Regel nicht erfüllt, wird das Paket fortan als Miss betrachtet.

- Sei $P \in \{0,1\}^n$ das eingehende Paket als Bitfeld.
- Sei $F \in \{0,1\}^n$ das Bitfeld gibt als Maske an welche Bits berücksichtigt werden sollen
- Die bitweise Filterung erfolgt durch eine Konjunktion

$$P_{neu} = P \wedge F = (P_1 \wedge F_1, P_2 \wedge F_2, \ldots, P_n \wedge F_n)$$

- Sei $R \in \{0,1\}^n$ das Bitmuster der Rule. Ein Paket wird als *Match* klassifiziert, wenn gilt:

$$P_{neu} = R$$

- Sonst wird das Paket als *Miss* behandelt:

$$P_{neu} \neq R$$

In der Matching-Deklaration kann außerdem unterschieden werden, ob Pakete explizit oder implizit gematcht werden sollen. Damit kann festgelegt werden, ob wir wie beim expliziten Match das genaue Bitfeld aus dem Datenfluss extrahieren wollen. Alternativ können wir mittels des impliziten Matchens Muster aus den Bitfeldern extrahieren. Sogleich können mehrere Bitfelder extrahiert werden, sofern diese die deklarierten Muster enthalten.

Sobald Bitfelder allgemein verarbeitet werden sollen, so muss dies mithilfe eines C Structs klar definiert werden. Dazu werden alle Bits des gewünschten Bitfeldes beim Definieren der Datenstruktur zunächst auf 1 gesetzt. In DOCA-Flow sieht eine Filter-Implementierung, die alle IP-Adressen in Gerade- und Ungerade-Klassen teilt, wie folgt aus:

```
match.outer.ip4.src_ip = BE_IPV4_ADDR(255, 255, 255, 255);
match_mask.outer.ip4.src_ip = BE_IPV4_ADDR(0, 0, 0, 1);
```

Zunächst wird durch das vollständige auf 1 Setzen des Bitfeldes der Source-IP, der Hardware signalisiert, dass nun ein implizites Matching auf eine Klasse von IP-Adressen erfolgen

soll. Somit ist die Match-Maske aktiviert. Dies bedeutet, dass wir nun nicht auf exakte Bitfelder matchen, sondern mittels der Match-Maske Bitmuster deklarieren können, die wir dann in den Entries matchen können. Nur allein die Match-Maske in der Pipe führt noch nicht zu Paketen, die gematcht werden. Es bedarf in der gezeigten Konfiguration immer noch eines speziellen Matches im Entry wie folgt:

```
match.outer.ip4.src_ip = BE_IPV4_ADDR(0, 0, 0, 1);
```

Die Kombination aus Pipe-Match und Entry-Match würde nun alle Pakete dem Entry zuordnen, die eine IP-Adresse haben, deren Quell-IP am letzten Bit eine 1 gesetzt hat.

4.3.4 DOCA Treiber

Damit die entsprechenden Hardwareeinheiten vom ARM-SmartNIC-System aus betrieben werden können, hat NVidia eigens bestimmte Treiber entwickelt. Dazu zählt unter anderem auch eine DPDK-Implementierung. DPDK steht für Data Plane Development Kit und ist ein von der Linux Foundation verwaltetes Open Source Projekt. Es wird verwendet, um Kernelprozesse der Datenverarbeitung aus dem Linuxkernel in den Userspace zu verlagern, um dort eine Verarbeitung vorzunehmen. DOCA Flow nutzt somit DPDK, um einige Pakete von dem Data Accelerator auf den ARM-Kern in den Userspace zu bringen, da DOCA Applikationen im Userspace ausgeführt werden.

Außerdem hat NVidia in DOCA andere Treiber entwickelt, wie beispielsweise Virtio-FS, mithilfe derer eine kompakte Integration von Dateisystemen in die BlueField-Systeme vorgenommen werden kann. Ein denkbarer Einsatzzweck wäre die dynamische Anbindung von Persistent Volumes in einem Kubernetes-Cluster, sofern sich die Volumes auf einem anderen Knoten befinden als der Pod, der versucht, das entsprechende Persistent Volume zu claimen.

4.3.5 Anwendungen

DOCA stellt eine Reihe von bereits implementierten Applikationen bereit, die eine Reihe von denkbaren Anwendungszwecken von DOCA präsentieren sollen. Im Rahmen dieser Arbeit wurden drei dieser Applikationen näher betrachtet.

So ist in einem Flyer auch von einem Load Balancer die Rede. Allerdings war dieser nicht auffindbar.

Firewall

Hier hat NVidia eine beispielhafte Firewall-Applikation entwickelt, deren Sinn und Zweck es ist, mit Hilfe von gRPCs zwischen Host und SmartNIC dynamisch Hardwareregeln hinzuzufügen bzw. diese wieder zu entfernen. Dabei wird der Host von dieser Aufgabe entlastet und die Pakete werden nicht erst auf dem Zielsystem, sondern bereits auf deren Netzwerkkarten verworfen.

GPU Packet Processing

Sollte im Hostsystem zusätzlich zur BlueField auch noch eine GPU verbaut sein, so bietet DOCA eine Applikation an, mit deren Hilfe Pakete von der GPU verarbeitet werden können. Das könnte beispielsweise für Anwendungen sehr nützlich sein, in denen nicht nur Header-Bitfelder verarbeitet werden, sondern auch die Payloads der Pakete verarbeitet werden sollen. Dazu eignen sich die Vektoreinheiten einer Grafikkarte gut, da diese auf hochdimensionale Vektorräume ausgelegt sind.

Switch

Die Kommunikation zwischen Host-System und der DPU wird mittels verschiedener Hardware-Networking-Interfaces angeboten. Bei der Switch-Applikation wird der Netzwerkverkehr auf der DPU in verschiedene Teilverkehre gespaltet und dann auf mehrere dieser Networking-Interfaces weitergegeben. Dadurch wird abermals das Hostsystem entlastet, da die spezielle Fallunterscheidung eines jeden Pakets nicht mehr vom Prozessor des Hosts übernommen werden muss, sondern bereits auf der Netzwerkkarte passiert. Außerdem bietet die SmartNIC-Architektur zusätzlich die Möglichkeit, bestimmte Hardwarebeschleuniger für diesen Einsatzzweck zu verwenden.

5 Netzwerklastverteilung

Lastverteilung spielt in der Computerwissenschaft eine sehr wichtige Rolle. Das Problem wird immer dann interessant, wenn ein Computersystem an seine Grenzen der Bearbeitbarkeit gebracht wird. So wird auch bei Mehrkernprozessoren ein Algorithmus derart aufgeteilt, dass jeder Kern ein Teilproblem löst, welches nach Fertigstellung des Teilproblems mit den Ergebnissen der anderen Kerne zusammengesetzt wird. So lässt sich in vielen Fällen ein sehr bedeutender Speedup für bestimmte Algorithmen erreichen, indem der Algorithmus auf mehrere Recheneinheiten verteilt wird. [21]

Betrachten wir nun Computerarchitekturen in Netzwerkumgebungen, so bilden wir eine Abstraktionsebene über den Mehrkernprozessoren. Nun ist es nicht mehr nur von Interesse, wie ein einzelner Prozessor das theoretische Problem bearbeitet, sondern es muss vielmehr betrachtet werden, wie sich besagtes Problem in einem Netzwerk aus Computern verhält. Mit dem Aufkommen des zivil und populär verfügbaren Internetanschlusses in den 90er Jahren ist vor allem die netzwerkbasierte Lastverteilung relevant. Die Kapazität eines einzelnen Computers, die Anfragen von Nutzern aus dem Internet zu verarbeiten, ist schnell erreichbar, sobald die Nachfrage auf Nutzerseite groß ist. So entstanden alsbald in den 90ern Ideen, wie es dennoch möglich ist, hochskalierende Anfrageraten zu beantworten.

Da im Rahmen dieser Arbeit ein L3/L4-Lastverteiler implementiert wird, werden im Folgenden die historischen wie aktuellen Ansätze für Lastverteilung auf diesen Schichten behandelt.

5.1 Entstehung

Wie bereits erwähnt, ist mit dem Aufkommen des Internets parallel das Interesse an Möglichkeiten der Lastverteilung gestiegen. Noch bevor dazu dedizierte Maschinen oder Anwendungen entwickelt wurden, gab es Ansätze, das Problem zu lösen. Dabei wurde der Aufbau des DNS-Systems genutzt. Wenn eine bestimmte URL von einem Client-Rechner angefordert wird, so wird zunächst eine Anfrage, in der die gesuchte URL steht, an einen DNS-Server geschickt. Dieser schlägt daraufhin nach, ob ihm die URL bekannt ist. Sollte dies der Fall sein, so schickt er ein Paket mit der entsprechenden IP-Adresse zurück an den Client-Rechner. [22] Dabei lässt sich nun auf ganz einfache Art und Weise die Last verteilen. Der DNS-Server hat, wie in Abbildung 5.1 zu sehen, nicht nur eine IP-Adresse für einen Server hinterlegt, sondern mehrere, aus denen er nun aus einem eigens dafür implementierten Algorithmus wählen kann, welche IP an den Client zurückgesendet wird. So konnte damals bereits mit einfachen Mitteln Last verteilt werden. DNS-Lastverteiler entscheiden weder nach MAC- noch nach IP-Adresse, an welche Backends weitergeleitet wird. Nach der OSI-Schichtenklassifizierung zählen DNS-Lastverteiler also am ehesten zu L7, lassen sich aber schwer einordnen.

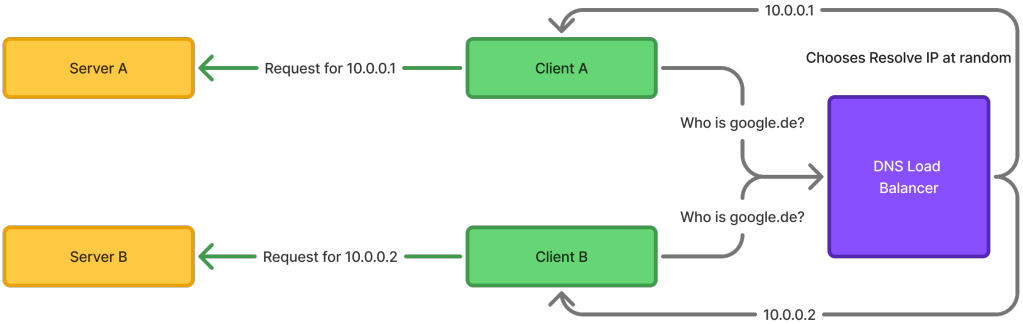

Abbildung 5.1: Aufbau eines DNS Loadbalancers

Allerdings bringt dieser Ansatz natürlich eine Reihe von Problemen mit sich. So kann der DNS-Server den aktuellen Status der Backend-Server nicht kennen. Sollte einer der Backend-Server offline sein, so wird weiterhin seine IP an die Clients gesendet und somit als funktionierendes Backend propagiert. Außerdem lässt ein DNS-Server keine intelligente Lastverteilung zu. Bei dieser verbesserten Form der Lastverteilung werden die Prozessorauslastung, Speichernutzung oder anderweitige Ressourcenstatistiken verwendet, um eine Entscheidung darüber zu fällen, welches Backend noch weitere Anfragen bearbeiten kann. Außerdem müsste der Client den entsprechenden DNS-Server standardmäßig konfiguriert haben. Dies ist aber nicht zwingend gewährleistet.

5.2 Hardwarelastverteiler (L2 bis L7)

Zum Ende der 90er Jahre hin zum Beginn der 2000er entwickelten eine Reihe von Herstellern hardwarebasierte Lastverteilungslösungen. Nennenswerte Hersteller sind hierbei F5-Networks mit **BIG-IP** (siehe Abbildung 5.2) aber auch Cisco oder Citrix. Bei diesen Geräten kamen auch erstmalig ASICs zum Einsatz. Grund dafür war die Tatsache, dass damalige Prozessoren bei weitem nicht schnell genug waren, um große Datenströme zu verarbeiten. Außerdem boten die damaligen Geräte zusätzliche Funktionalitäten an. Darunter waren grafische Nutzeroberflächen oder integrierte Backend Health Checks. [23] Die damaligen ASICs fokussierten sich damals vor allem auf L4-Paket-Forwarding. Außerdem wurden derartige Geräte meist als eine Art Komplettlösung für sämtliche Aufgaben des Datenverkehrs vermarktet. So wurde auf selbigen Geräten meistens eine Firewall mit angeboten, wodurch auch sicherheitstechnische Funktionen wie Distributed Denial of Service Attacken bekämpft werden konnten. All diese Funktionen konnten mithilfe von Logging untersucht und analysiert werden. Außerdem konnten einige Hardwarelastverteiler auch an

Abbildung 5.2: BIG-IP von F5-Networks [39]

Anwendungen angepasst werden, sodass zusätzlich eine L7-Lastverteilung möglich wurde.

5.3 Softwarelastverteiler L3/L4 bis L7

Softwarelastverteiler unterscheiden sich deutlich von den Hardwarelastverteilern in dem Punkt, dass bei softwarebasierter Lastverteilung keine spezialisierte Hardware in Form von ASICs oder FPGAs zum Einsatz kommt. Es werden lediglich Programme auf Servern ausgeführt, deren Funktion es ist, den Netzwerkverkehr zu steuern. Wie im vorherigen Absatz beschrieben, ist dies nur aufgrund der Entwicklungen im Bereich der Hardware der letzten zwei Jahrzehnte möglich geworden. Während dieser Zeit wurden gängige Prozessoren nicht nur um Rechenkerne erweitert, sondern erhielten außerdem größeren Cache und schnellere Anbindungen an sonstige Ein- und Ausgabeeinheiten. Außerdem lassen sich softwarebasierte Lastverteiler deutlich besser in moderne, meist VM-betriebene Netzwerkumgebungen integrieren. Die Natur eines anwendungsbasierten Lastverteilers bietet die Möglichkeit, genaue anwendungsfallspezifische Modifikationen vorzunehmen, womit eine kompakte Integration in ein System vorgenommen werden kann. [24] So entstanden eine Menge von nennenswerten Anwendungen, die sich eine Lösung von softwarebasierter Lastverteilung zum Ziel gesetzt haben. Die wohl bekanntesten aktuellen Vertreter dieser Klasse sind Programme wie HAProxy, Nginx und mod_proxy von Apache. [20] Alle diese sind allerdings Lastverteiler der Klasse L7 und somit Userspace-Lastverteiler. Außerdem ist ebenfalls an der Universität Potsdam im Rahmen der Masterarbeit von Phillip Ungrund der Katran-Lastverteiler in einem ähnlichen Testszenario untersucht worden. Diese Messreihe ist sehr gut mit den Folgenden vergleichbar, da in Phillip Ungrunds Messungen ebenfalls DNS-Pakete verwendet wurden [32].

5.4 Kubernetes und Userspace Load Balancer

Kubernetes ist eine moderne Container-Runtime, welche im Gegensatz zu klassischem Docker oder Podman nicht nur auf einem Rechner laufen kann, sondern gleich eine Netzwerkabstraktion mit sich bringt. In dieser werden mehrere Computer zu einem Kubernetes-Cluster zusammengefasst und können mittels dedizierter Control-Plane-Knoten kontrolliert werden. In solch einer Umgebung wird fast vollständig über REST-APIs kommuniziert. Anwendungen werden in solch einem Cluster in feingranulare Microservices heruntergebro-

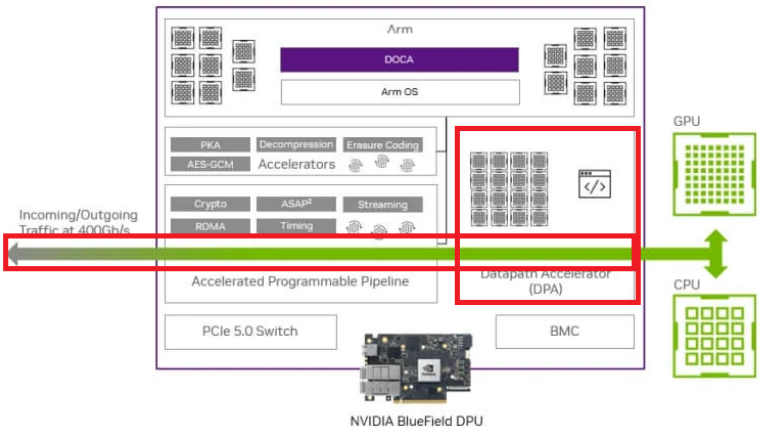

Abbildung 5.3: Für Hardwarelastverteiler relevante Bereich auf einer DPU [36]

chen. Ziel eines solchen Ansatzes ist es, die Teile eines komplexen Systems einzeln wartbar zu machen, aber auch Skalierfähigkeit zu erhalten. Grundsätzlich wird in Kubernetes-Clustern stets empfohlen, den einzelnen Microservices möglichst wenig State zu geben. Damit soll garantiert werden, dass die Provisionierung von Kubernetes übernommen werden kann. Dinge wie Replikas und redundante Anwendungen können so von Kubernetes bereitgestellt werden. [25]

In besagter Kubernetes-Architektur werden Ressourcen typisiert. So werden Container in Ressourcen vom Typ **Pod** ausgeführt. Diese sind meistens ebenfalls ein Teil einer abstrakteren Ressource namens **Deployment**, in der mehrere Pods zusammengefasst werden können. Damit nun ein solches Deployment von außerhalb des Kubernetes Clusters erreichbar ist, muss eine Ressource vom Typ **Service** definiert werden. Solch ein Service kann wiederum unterschiedliche Arten haben. Ist die Art des Services **LoadBalancer** so wird an einen eventuellen Cloud Provider wie GKE (Google), AWS (Amazon) oder Azure (Microsoft) kommuniziert, dass eine Lastverteilungs-IP-Adresse bereitgestellt werden soll, die auf den verknüpften Service zeigen soll. Dieser Service zeigt wiederum auf das Deployment, in dem entsprechend der gesuchte Endpunkt für die Kommunikation liegt. Der im Rahmen dieser Arbeit implementierte Load Balancer eignet sich in einem solchen Szenario sehr gut, da er auf der Netzwerkkarte des Providers in den für die Lastverteilung bestimmten Servern ausgeführt werden kann.

5.5 Lastverteilung auf einer Data Processing Unit (DPU)

DPUs bieten eine weitere Plattform für Hardwarelastverteiler, die zu den jüngsten Vertretern dieser Klasse zählen. Wie in Abbildung 5.3 zu sehen, werden bei dieser Art von Architektur die Netzwerkkarten mit ihren entsprechenden (hier rot markierten) Hardwarebeschleunigern selbst zur Lastverteilung verwendet. So kann gänzlich auf einen separaten Lastverteilungsknoten verzichtet werden.

6 XenoFlow - ein L3/L4 Loadbalancer

Ziel dieser Bachelorarbeit war es, einen möglichst hardwarebeschleunigten Lastverteiler zu entwickeln. Dazu wurde das DOCA Flow Framework verwendet, da die zur Verfügung stehende Netzwerkkarte eine BlueField-3 ist. Im folgenden Kapitel wird ein Überblick über die Entwicklung mit DOCA Flow gegeben und es werden die Kernelemente des XenoFlow-Lastverteilers vorgestellt.

XenoFlow ist nicht für Produktionsumgebungen gedacht, sondern soll vielmehr ein Proof-of-Concept darstellen und einen speziellen Anwendungsfall demonstrieren. XenoFlow ist aktuell auch nur in der Lage, UDP-Pakete zu verarbeiten. Diese Entscheidung ist damit zu begründen, dass beispielsweise eine TCP-Verbindung mehr Umstände beinhaltet, die berücksichtigt werden müssten. Zunächst soll aber nur die Machbarkeit und die allgemeine Performance beurteilt werden. Der komplette Quellcode von XenoFlow ist unter der MIT veröffentlicht und unter den weiter unten angegebenen Git-Repositories verfügbar.

6.1 Entwicklung

NVidia stellt eine Menge von Beispielapplikationen für die verschiedenen DOCA-Frameworks und -Libraries zur Verfügung. Der erste Teil der Entwicklung dieses Lastverteilers bestand also darin, eine Wissensbasis für DOCA und hier im Speziellen DOCA Flow aufzubauen. Dabei fielen allerdings gerade zu Beginn der Entwicklung viele Probleme auf. Das wohl größte Problem ist, dass DOCA und damit auch DOCA Flow nur mit einer sehr schlechten Dokumentation ausgestattet sind. So werden die meisten Teilbereiche zwar angesprochen, aber leider meist nicht im Detail erläutert. Außerdem finden sich häufig widersprüchliche Aussagen in der Dokumentation. Zuweilen werden auch Abkürzungen verwendet, die nirgends klar definiert werden und somit den Leser in völligem Unwissen darüber lassen, was sie zu bedeuten haben sollen. Oftmals gewinnt man als Leser den Eindruck, es handele sich um eine relativ diffus zusammengeworfene Wissensbasis, die keiner Qualitätskontrolle unterlaufen ist. Es schienen mehrere Entwicklungsteams an dem Verfassen der Dokumentation beteiligt gewesen zu sein, da oftmals stark unterschiedliche Programmierstile verwendet wurden. Dies führt abermals zur Verwirrung auf Seiten des Lesers. Dazu sei erwähnt, dass etwaige Enterprise-Produkte wie in diesem Fall BlueField so vermarktet werden, dass die eigentliche Hardware nur einen Teil der Dienstleistung umfasst. Oftmals werden dazu noch Schulungen gebucht, die das beteiligte Entwicklungspersonal in der Benutzung der Plattform unterweisen sollen.

Diese Möglichkeiten sind aus finanziellen als auch personaltechnischen Gründen nicht an einer deutschen Hochschule gegeben. Somit ist es dem auf dem Gebiet arbeitenden Personal selbst überlassen, Informationen auf dem Gebiet zu gewinnen. Auch das NVidia-Forum, welches ein spezielles Unterforum für die BlueField-Familie besitzt, stellte sich als nicht besonders informativ heraus. Bei einer Anfrage wird entweder nicht geantwortet oder auf

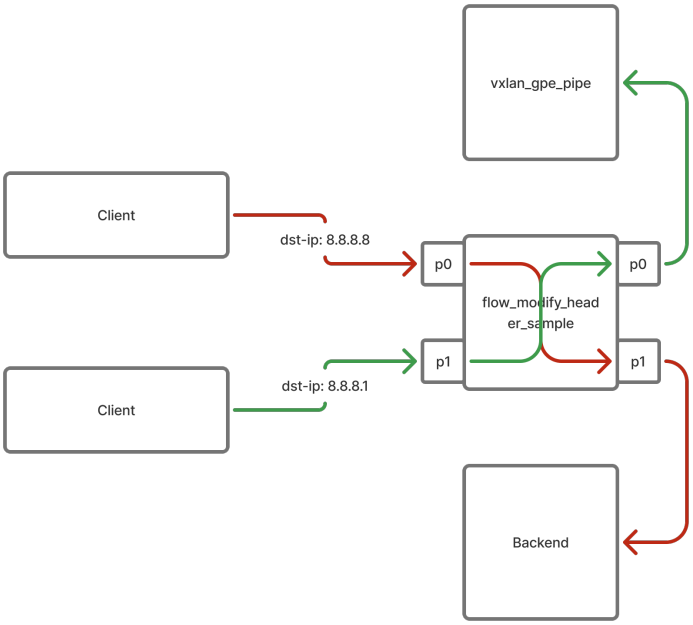

Abbildung 6.1: Aufbau des Flow-Modify-Header-Sample

eine generische Support-E-Mail-Adresse verwiesen. Wendet man sich an besagte Adresse, soll ein Zertifikat vorgelegt werden, ehe man mit einem tatsächlichen Entwickler Kontakt aufnehmen kann.
Bei den ersten Experimenten auf der BlueField-3 war es zunächst nicht möglich, den Netzwerkverkehr vollständig auf der spezialisierten Hardware zu verarbeiten. Sobald Paketlast entstand, war die Performance an einen einzigen ARM-Kern gebunden und es konnten nur sehr schlechte Verarbeitungsraten erzielt werden. Erst als der Open vSwitch auf der BlueField selbst wie folgt konfiguriert wurde

```
ovs-ofctl add-flow br0 "in_port=1,actions=output:2"
```

verschwand die Last auf den ARM-Kernen und der Verkehr war somit komplett hardwarebeschleunigt.

6.1.1 Modify Header und Shared Counter

Zu Beginn der Entwicklung wurde auf der Beispielapplikation flow_modify_header_sample.c aufgebaut. Das besagte Beispielprojekt beschreibt eine Anwendung, in der Netzwerkverkehr auf das bestimmte Destination-IP-Feld mit der IP 8.8.8.8 gematcht und daraufhin die MAC-Adresse mittels Action geändert wird. Der Aufbau der wichtigen Kompo-

nenten ist in Abbildung 6.1 zu sehen. Dieser Ansatz ist für die Implementierung eines Load Balancers von großem Interesse. Grund dafür ist, dass ein Netzwerkpaket so manipuliert werden kann, dass es den mit der MAC-Adresse spezifizierten Rechner erreicht. Dort wird dann eine Modifikation der MAC-Adresse vorgenommen, damit das Paket an ein dediziertes Backend weitergeleitet wird. Zusätzlich wurde noch die Beispielapplikation flow_shared_counter_sample.c untersucht. In dieser wird ein Beispiel gegeben, wie von der BlueField erfasste Monitoring-Daten abgerufen werden können, um Informationen über den aktuellen Datenverkehr erhalten zu können. Dies ist, sofern es wirklich Hardwareeinheiten sind, nicht trivial, da es keinen Speicherbereich oder Ähnliches gibt, auf den zugegriffen werden kann, um Daten zu erhalten. DOCA stellt dafür eine Funktion bereit:

```
int doca_flow_shared_resources_query(DOCA_FLOW_SHARED_RESOURCE_COUNTER,
                                     shared_counter_ids,
                                     query_results_array,
                                     nb_ports);
```

Laut der Dokumentation können so mehrere Entries auf den Counter zugreifen bzw. diesen erhöhen. Der gesamte Prozess ist somit vergleichbar mit Thread-Safety bei parallelen Programmen, abgebildet in der DOCA Flow-Semantik mit Flow Entries.

In Abbildung 6.1 ist dargestellt, wie zwei Clients jeweils ein Paket mittels genauer MAC-Adresse an die modify header Applikation schicken. Der rote Pfad beschreibt dabei ein Paket, welches von der ersten Pipe verarbeitet wird. Daraufhin wird die entsprechende Source MAC-Adresse bearbeitet. Der grüne Pfad beschreibt hingegen ein Paket, welches nicht von der ersten Pipe und deren Entries verarbeitet wurde. Es wird somit als Miss klassifiziert und in der spezifischen Applikation in eine andere Pipe weitergeleitet. Diese hat im Beispiel nicht für diese Arbeit relevante weitere Operationen in bestimmten Entries definiert. Alle von einem bestimmten Hardwareport der BlueField verarbeiteten Pakete werden an den jeweils anderen Hardwareport weitergeleitet.

6.1.2 Entwurf des Lastverteilers

Im Anschluss an diverse Experimente zum Verständnis von DOCA Flow mithilfe der Beispielapplikationen wurde das Grundgerüst für XenoFlow implementiert. Dazu wurde erstmalig überlegt, wie der Verkehr in verschiedene Teilnetzverkehre aufgeteilt wird, damit dieser dann auf mehrere Backends verteilt werden kann. Zunächst sollte in Anlehnung an die Masterarbeit von Phillip Ungrund ein Algorithmus verwendet werden, der die Source-Adresse im IP Header analysiert und anhand dieser eine Prüfsumme errechnet. Diese Hashsumme soll dann überprüft werden, ob sie bereits bekannt ist. Sollte dies der Fall sein, so wird sie an die gleiche Backendadresse weitergeleitet. Wenn nicht, so wird ein Backend zufällig gewählt und ein entsprechender Eintrag in einer Datenstruktur für späteren Netzwerkverkehr angelegt. Dieser Ansatz wurde allerdings im Laufe der weiteren Gespräche in der Gruppe verworfen. Grund dafür ist, dass ein solcher Algorithmus nicht hardwarebeschleunigt laufen könnte. Er müsste von einer der spezialisierten Hardwareeinheiten ausgeführt werden. Diese sind allerdings in ihrer Funktionalität nicht frei programmierbar. Somit müsste jedes unbekannte Datenpaket von einer Hashfunktion analysiert werden,

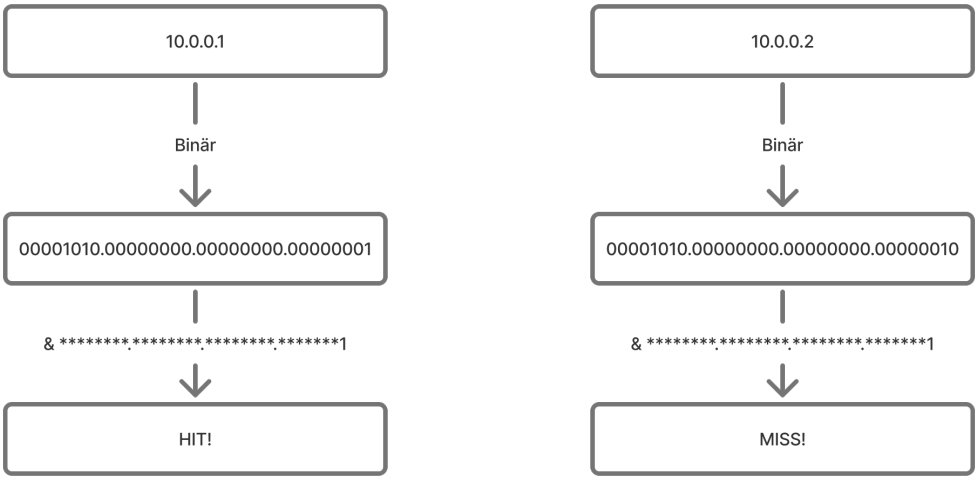

Abbildung 6.2: XenoFlow Matching Algorithmus

welche auf dem ARM-Prozessor ausgeführt wird. Damit würde abermals die Idee der vollständigen Hardwarebeschleunigung verloren gehen und die Performance des Lastverteilers wäre an den Prozessorkern gebunden. Daher wurde zunächst die Untersuchung der tatsächlichen Performance der BlueField-3 Verarbeitung in den Vordergrund gestellt, womit ein wesentlich simplerer, aber deutlich leichter zu implementierender Algorithmus entwickelt wurde. Auf der Ebene der Bitfelder wird die Source-Adresse im IP Header verwendet und mittels des Filter-Matchings überprüft, ob die letzten Bits einer Client-IP einem bestimmten Muster entsprechen. Die Lastverteilung von XenoFlow findet auf der Ebene der IP-Adresse statt, weshalb es sich um einen L3/L4-Lastverteiler handelt. In Abhängigkeit davon, in welcher Klasse ein Datenpaket erfasst wurde, wird nun entsprechend der Liste an verfügbaren Backend-Servern die Ziel-MAC-Adresse des Paketes bearbeitet. Abschließend verlässt das Paket auf dem Port-1 der BlueField-3 die Verarbeitung und wird vom Switch an das entsprechende Backend weitergeleitet. In den folgenden Versuchen wurde Port-0 für eingehenden Netzwerkverkehr und Port-1 für ausgehenden Netzwerkverkehr verwendet. In Abbildung 6.2 ist ein beispielhafter Aufbau angegeben. Hierbei werden zwei Source-Adressen untersucht. Bei der Adresse 10.0.0.1 ist das letzte Bit der IP gesetzt. Wird also daraufhin der Filter mittels Und-Operator angewendet, so führt diese Adresse zu einem Hit und wird in die entsprechende Pipe eingefügt bzw. von dem entsprechenden Entry verarbeitet.

Bei einer IP-Adresse, bei der allerdings das letzte Bit nicht gesetzt ist, wie es bei 10.0.0.2 der Fall ist, wird der Und-Filter das Paket als Miss interpretieren. Der beschriebene Al-

gorithmus beschreibt hier natürlich nur einen Teil, zeigt aber in seinem Aufbau, wie sich anhand leichter Filter zwei Klassen von IP-Adressen erstellen lassen. Genauer wird in gerade und ungerade IP-Adressen im letzten der 4 Adressfelder unterschieden. Der große Vorteil dieser Implementierung ist es, dass der Filter vollständig hardwarebeschleunigt arbeiten kann. Es muss somit keinerlei Algorithmik oder Ähnliches durchlaufen werden, um die IP-Klassen zu befüllen. Dies sollte sich in den späteren Versuchen anhand geringer Latenzen deutlich machen.

6.1.3 Definition von Backends

Backends, welche in der Lage sind, Clientanfragen zu bearbeiten und dementsprechend von dem Lastverteiler als valide Endpunkte für eingehenden Verkehr angesehen werden sollen, können in XenoFlow deklarativ definiert werden. Hintergrund dafür ist, dass moderne Systeme wie Kubernetes ebenfalls auf einen deklarativen Konfigurationsstil setzen. Dazu wird die Formatierung **json** verwendet.

So können Backends für XenoFlow definiert werden, indem wie folgt JSON-Dateien angelegt werden:

```
{
    "backends": [
        {
            "name": "fips2",
            "mac_address": "A0:88:C2:B5:F4:5A"
        },
        {
            "name": "fips3",
            "mac_address": "E8:EB:D3:9C:71:AC"
        }
    ]
}
```

Dabei spielen die Namen der jeweiligen Backends nur in der Ausgabe von XenoFlow eine Rolle und sind mehr dazu gedacht, bei späteren Weiterentwicklungen von XenoFlow verwendet zu werden. Von größerem Interesse sind die MAC-Adressen, die mit einem jeweiligen Backend assoziiert werden. Diese werden folgend im Lastverteiler verwendet, um eingehende Pakete an diese Adresse weiterzuleiten. Hierbei sei auch erwähnt, dass, ähnlich wie bei DNS-basierten Lastverteilern, im aktuellen Implementierungsstatus nicht sichergestellt ist, ob das jeweilige Backend überhaupt in der Lage ist, Traffic zu verarbeiten. Dies ist allerdings ein Feature, welches sich ebenfalls zu einem späteren Zeitpunkt und unabhängig von dieser Arbeit dazu implementieren lassen würde. Außerdem wäre eine Erweiterung denkbar, die **yaml** als zusätzliche Markup-Sprache akzeptiert, da yaml wesentlich häufiger in Kontexten wie Kubernetes verwendet wird. Damit wäre XenoFlow ein wenig mehr in line mit modernen Cloudsystemen und deren syntaktischen Entwicklungen.
Die nun definierten Backends modifizieren die entsprechenden Hardware-Filterregeln. Würde eine Erweiterung hinsichtlich der Health Checks sowie anderer Funktionalitäten wie

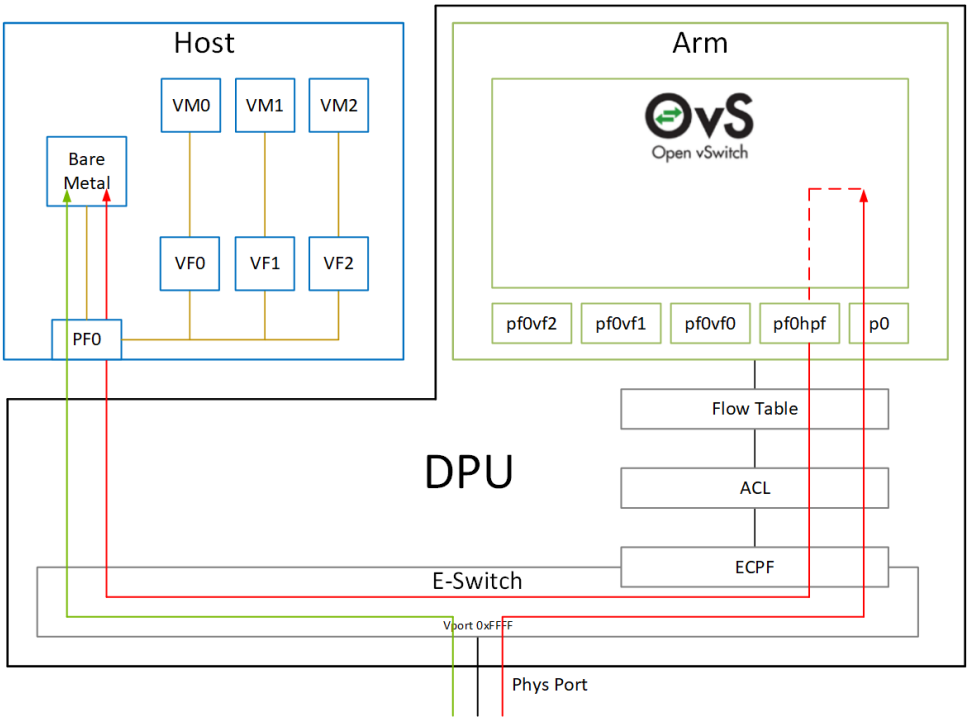

Abbildung 6.3: Open vSwitch auf der BlueField-3

Logging erfolgen, so ist es dringend angeraten, die allgemeine Architektur der Backendwahl zu modifizieren. Im Rahmen dieser Arbeit wurde die Architektur allerdings bewusst einfach gehalten, um die tatsächliche Leistung der BlueField zu beurteilen und nicht die der entsprechenden Architektur.

6.1.4 Open vSwitch

Die Bluefield verwendet, wie in den vorherigen Kapiteln beschrieben, virtuelle Netzwerkinterfaces, um zwischen dem Host und den ARM-Kernen zu kommunizieren, aber auch, um den Verkehr zwischen den Hardwareports zu steuern. Dazu verwendet NVidia Open vSwitch. Open vSwitch ist eine Applikation, um die Funktionalität eines Switches, der Multilayer arbeiten kann, per Software in Linux abzubilden. Das Projekt wird von der Linux Foundation entwickelt und ist komplett quelloffen unter der Apache-Lizenz 2.0 veröffentlicht. Laut NVidia werden Switch-Regeln, die in Open vSwitch angelegt werden, mittels der ASAP²-Technologie auf die Hardwarebeschleuniger ausgelagert. Dazu ist in Abbildung 6.3 ein Datenpfad von zwei unterschiedlichen Paketen abgezeichnet. Der grüne Pfad zeigt ein Paket, welches direkt vom Switch an den Host hochgereicht wird und dort unter dem Interface PF0 abgeholt werden kann. Das Paket des roten Pfades hingegen durchläuft zunächst den ARM-Kern mit den entsprechenden Interfaces. Unsere Lastverteilung allerdings soll keinen der beiden angegebenen Pfade durchlaufen, sondern lediglich auf den Hardwarebeschleunigern ausgeführt werden. Somit muss allerdings auch deklariert werden, dass eingehender Traffic direkt wieder aus dem entsprechenden anderen Hardwareport ausgegeben

wird. Dies ist dank der Erweiterung von OVS von NVidia möglich. Diese Erweiterung trägt den Namen NVIDIA's DOCA-OVS. Im Speziellen soll DOCA-OVS API-Aufrufe ergänzen, die ermöglichen, dass die Hardware entsprechend mit den definierten Regeln arbeiten kann.

6.2 Quellcode

Im folgenden Abschnitt wird ein genauer Überblick über den XenoFlow Lastverteiler gegeben. Es werden die verschiedenen C-Strukturen und Funktionen, die intrinsisch von NVidia bereitgestellt werden, vorgestellt.

6.2.1 main()

Zu jeder Flow-Applikation gehören nicht nur die entsprechenden Datenstrukturen zur Definition von Pipes und Entries, sondern es müssen auch Konfigurationen allgemeinerer Natur getroffen werden. Diese werden wie üblich in einer _main.c zusammengefasst. In dieser Quellcode-Datei liegt außerdem die eigentliche main-Funktion, die zu Beginn eines jeden C-Programms ausgeführt wird:

```
int main(int argc, char **argv)
{
        doca_error_t result;
        struct doca_log_backend *sdk_log;
        int exit_status = EXIT_FAILURE;
        struct application_dpdk_config dpdk_config = {
                .port_config.nb_ports = 2,
                .port_config.nb_queues = 4,
                .port_config.nb_hairpin_q = 2,
        };
```

Zunächst werden hier einige Datenstrukturen deklariert. Interessant sind hierbei die `doca_log_backend` und `application_dpdk_config` Strukturen.

Das Logbackend wird verwendet, um festzulegen, wie und welches Loglevel erfasst werden soll. Hierzu wird mit den intrinsischen Funktionsaufrufen

```
result = doca_log_backend_create_standard();
if (result != DOCA_SUCCESS)
    goto sample_exit;
result = doca_log_backend_create_with_file_sdk(stderr, &sdk_log);
if (result != DOCA_SUCCESS)
    goto sample_exit;
result = doca_log_backend_set_sdk_level(sdk_log, DOCA_LOG_LEVEL_WARNING);
if (result != DOCA_SUCCESS)
    goto sample_exit;
```

die entsprechende Struktur initialisiert und `stderr` als Output-Zeiger übergeben. Hier könnten theoretisch auch andere Ziele für die entsprechenden Lognachrichten festgelegt

werden. Außerdem wird das minimale Loglevel festgelegt. Lognachrichten, die unter diesem Loglevel liegen, werden nicht angezeigt. Es haben sich hierbei folgende Loglevel entwickelt, die auch in DOCA verwendet werden können:

- DEBUG - sehr verbose und ausführliche Informationen zum Auffinden von Fehlern im Betrieb
- INFO - allgemeine Informationen über den Programmstatus
- WARNING - Warnungen über eventuell aufkommende Fehler
- ERROR - direkte Fehlermeldungen

Eine klassische Lognachricht des Levels INFO sieht wie folgt aus:

```
DOCA_LOG_INFO("Starting the load balancer");
```

Als nächstes wird DPDK initialisiert. Dazu wird abermals mittels einer DOCA-intrinsischen Funktion die zuvor erstellte Datenstruktur übergeben. Außerdem wird der Name, der folgend als ID für die laufende DOCA-Applikation verwendet wird, festgelegt:

```
doca_argp_init("doca_flow_lb", NULL);

doca_argp_set_dpdk_program(dpdk_init);

doca_argp_start(argc, argv);
```

Mittels des letzten Funktionsaufrufs werden die Parameter der Konsole mit an DOCA übergeben. Dies ist vor allem dazu gebräuchlich, um zu deklarieren, über welches PCI-Interface mit der BlueField kommuniziert werden soll, da es möglich ist, mehrere BlueFields in einem Host zu verbauen. Zuletzt wird nun unsere eigentliche Applikation ausgeführt:

```
xeno_flow(dpdk_config.port_config.nb_queues);
```

Dabei wird die konkrete Anzahl an DPDK-Queues angegeben, die zur Kommunikation mit dem ARM-BlueField-Host verwendet wird.

6.2.2 xeno_flow()

Es werden sämtliche Datenstrukturen deklariert, die im Folgenden von XenoFlow gebraucht werden:

```
int nb_ports = 1;
struct flow_resources resource = {1};
uint32_t nr_shared_resources[SHARED_RESOURCE_NUM_VALUES] = {0};
struct doca_flow_port *ports[2];
struct doca_dev *dev_arr[nb_ports];
struct doca_flow_pipe *udp_pipe;
int port_id = 0;
uint32_t shared_counter_ids[] = {0, 1};
```

```
struct doca_flow_resource_query query_results_array[nb_ports];
struct doca_flow_shared_resource_cfg cfg = {.domain = DOCA_FLOW_PIPE_DOMAIN_DEFAULT};
struct entries_status status;
int num_of_entries = 4;
doca_error_t result;
nr_shared_resources[DOCA_FLOW_SHARED_RESOURCE_COUNTER] = 2;
XenoFlowConfig *config = load_config();
```

Hier wird definiert, wie viele Ports von XenoFlow verwendet werden sollen - *allerdings nur, wie viele später mit Pipes genutzt werden; es kann auf mehr Ports weitergeleitet werden* - und es werden die Strukturen für die späteren Pipedefinitionen festgelegt. Außerdem wird in der letzten Zeile die Konfiguration aus der `backends.json` geladen. Diese kann dann später vom Lastverteilungsalgorithmus verwendet werden. In DOCA Flow wird ein sehr expliziter Programmierstil gefordert. Es muss stets genau angegeben werden, wie viele Elemente von Strukturen an DOCA übergeben werden. So muss beispielsweise die Anzahl an Entries in den Pipes klar angegeben sein.
Nun können die eigentlichen Konfigurationsfunktionen aufgerufen werden. Es sei kurz erwähnt, dass die Flow-Beispielapplikationen meist das Errorhandling mittels eines `if` realisieren. Da dies zu einem enorm langen und teils unleserlichen Quelltext führt, wurde die Funktion `doca_try` implementiert, die im Wesentlichen nur den `return` des Funktionsaufrufs überprüft. Ist dieser != DOCA_SUCCESS, so wird eine Fehlermeldung ausgegeben. Damit konnte der Quellcode um ca. 100 Zeilen verkürzt werden.

```
DOCA_LOG_INFO("Number of backends: %d", config->numBackends);

doca_try(
    init_doca_flow(
        nb_queues,
        "vnf,hws",
        &resource,
        nr_shared_resources
    ), "Failed to init DOCA Flow", nb_ports, ports
);

doca_try(
    init_doca_flow_ports(
        2,
        ports,
        true,
        dev_arr
    ), "Failed to init DOCA ports", nb_ports, ports
);

doca_try(
    doca_flow_shared_resource_set_cfg(
        DOCA_FLOW_SHARED_RESOURCE_COUNTER,
```

```
        0,
        &cfg
    ), "Failed to configure shared counter to port", nb_ports, ports
);



doca_try(
    doca_flow_shared_resources_bind(
        DOCA_FLOW_SHARED_RESOURCE_COUNTER,
        &shared_counter_ids[0],
        1,
        ports[0]
    ), "Failed to bind shared counter to pipe", nb_ports, ports
);
```

Mittels `init_doca_flow()` wird DOCA Flow initialisiert. Somit wird sichergestellt, dass die API, die mit den Hardwareeinheiten kommuniziert, verfügbar ist und auf kommende Eingaben wartet. Selbiges gilt für `init_doca_flow_ports()`. Hier werden die entsprechenden beiden Hardwareports an der Netzwerkkarte angewiesen, dass nun Traffic erwartet werden kann, der mittels Steering-Regeln aus DOCA Flow kontrolliert wird. Anschließend werden mittels `doca_flow_shared_resources_bind()` die Counter-Ressourcen gebunden. Diese werden im späteren Programmablauf verwendet, um einen Livestatus über die verteilte Last zu erhalten. Diese Ressource muss abermals intrinsisch an DOCA Flow übergeben werden, damit das entsprechende Hardwarebackend angesprochen werden kann.

```
doca_try(
    create_root_pipe(
        ports[0],
        0,
        DOCA_FLOW_L4_TYPE_EXT_UDP,
        &udp_pipe
    ), "Failed to create pipe", nb_ports, ports
);

doca_try(
    add_shared_counter_pipe_entry(
        udp_pipe,
        DOCA_FLOW_L4_TYPE_EXT_UDP,
        shared_counter_ids[0],
        &status
    ), "Failed to add entry", nb_ports, ports
);

doca_try(
```

```
    doca_flow_entries_process(
        ports[0],
        0,
        DEFAULT_TIMEOUT_US,
        num_of_entries
    ), "Failed to process entries", nb_ports, ports
);
```

Nun werden die beiden entscheidenden Funktionen `create_root_pipe()` und `add_shared_counter_pipe_entry()` aufgerufen. Diese beinhalten die entsprechende Lastverteilungslogik. Zuletzt wird DOCA Flow angewiesen, die angelegten Entries zu verarbeiten. In diesem Schritt findet die eigentliche Hardwareauslagerung statt. Die Funktion dazu lautet `doca_flow_entries_process()`.

6.2.3 Pipe- und Entry-Erstellung

Die folgenden Quellcodeabschnitte wurden aufgrund der besseren Lesbarkeit leicht gekürzt. Im folgenden Code ist zunächst die Erstellung einer Pipe, die zunächst sämtlichen Netzwerkverkehr abfängt:

```
static doca_error_t create_root_pipe(
    struct doca_flow_port *port,
    int port_id,
    enum doca_flow_l4_type_ext out_l4_type,
    struct doca_flow_pipe **pipe
) {
    struct doca_flow_match match;
    struct doca_flow_match match_mask;
    struct doca_flow_monitor monitor;
    struct doca_flow_actions actions0, *actions_arr[2];
    struct doca_flow_fwd fwd, fwd_miss;
    struct doca_flow_pipe_cfg *pipe_cfg;

    match.outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match.outer.ip4.src_ip = BE_IPV4_ADDR(255, 255, 255, 255);
    match_mask.outer.ip4.src_ip = BE_IPV4_ADDR(0, 0, 0, 1);
    DOCA_LOG_INFO("%d", match_mask.outer.ip4.src_ip);

    SET_MAC_ADDR(
        actions0.outer.eth.dst_mac,
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff
    );
    SET_MAC_ADDR(
        actions0.outer.eth.src_mac,
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff
    );
```

```
    actions_arr[0] = &actions0;
    monitor.counter_type = DOCA_FLOW_RESOURCE_TYPE_SHARED;
    monitor.shared_counter.shared_counter_id = 0xffffffff;

    doca_flow_pipe_cfg_create(&pipe_cfg, port);
    set_flow_pipe_cfg(
        pipe_cfg,
        "SHARED_COUNTER_PIPE",
        DOCA_FLOW_PIPE_BASIC,
        true
    );
    doca_flow_pipe_cfg_set_match(pipe_cfg, &match, &match_mask);
    doca_flow_pipe_cfg_set_actions(pipe_cfg, actions_arr, NULL, NULL, 1);
    doca_flow_pipe_cfg_set_monitor(pipe_cfg, &monitor);

    fwd.type = DOCA_FLOW_FWD_PORT;
    fwd.port_id = 1;
    fwd_miss.type = DOCA_FLOW_FWD_DROP;

    doca_flow_pipe_create(pipe_cfg, &fwd, &fwd_miss, pipe);
}
```

Wie schon in Kapitel 3 dem Aufbau einer Pipe zu entnehmen ist, besteht eine Pipe grundsätzlich aus den Elementen **Match**, **Monitor**, **Action** und **Forward**. Diese werden mittels entsprechender DOCA Flow Structs deklariert und mit den entsprechenden Werten gefüllt. Hierzu sei erwähnt, dass damit alle Bits für das Matching oder eine spätere Action als aktiv geschaltet sein müssen. Sollen sie also im späteren Programmablauf gelesen oder modifiziert werden können, müssen die entsprechenden Bits in der Pipe-Deklaration als 1 gesetzt werden. Dazu sind z.B. die Counter-ID auf `0xffffffff` oder die src_mac auf `0xff, 0xff, 0xff, 0xff, 0xff, 0xff`. Mit der Erstellung der Pipe sollten die entsprechenden Bitfelder auf der Hardware der BlueField-3 aktiviert sein. Da wir einen impliziten Match forcieren wollen, muss außerdem eine zweite Filtermaske gesetzt werden, die bei Aufruf der intrinsischen Funktion `doca_flow_pipe_cfg_set_match()` mit übergeben wird. Außerdem müssen bereits die `doca_flow_actions` angegeben werden, die in den einzelnen Entries zur Anwendung kommen. Dabei wird nicht spezifiziert, was genau modifiziert werden soll, sondern nur, welche Bitfelder zur Modifikation freigegeben sind.

Zuletzt wird mittels der Funktion `doca_flow_pipe_create()` die entsprechende Pipe final angelegt. Ist dieser Funktionsaufruf ohne Fehlermeldung und erfolgreich, so ist die Hardwaresteering-Regel aktiv und der Traffic wird nun gemäß den angelegten Regeln verarbeitet.
Da mit der Pipe noch nicht die spezielle Lastverteilung eingebaut ist, sondern lediglich definiert wurde, welche Bitfelder verarbeitet werden sollen und welcher Traffic überhaupt abgegriffen werden soll, ist im folgenden Quelltext die Erstellung der beiden speziellen Entries angegeben:

```
static doca_error_t add_shared_counter_pipe_entry(
```

```
    struct doca_flow_pipe *pipe,
    enum doca_flow_l4_type_ext out_l4_type,
    uint32_t shared_counter_id,
    struct entries_status *status
) {
    /**
        ...
    **/

    monitor.shared_counter.shared_counter_id = shared_counter_id;

    match.outer.ip4.src_ip = BE_IPV4_ADDR(0, 0, 0, 1);
    actions.action_idx = 0;
    SET_MAC_ADDR(
        actions.outer.eth.dst_mac,
        0xe8, 0xeb, 0xd3, 0x9c, 0x71, 0xac
    );
    SET_MAC_ADDR(
        actions.outer.eth.src_mac,
        0xc4, 0x70, 0xbd, 0xa0, 0x56, 0xbd
    );
    doca_flow_pipe_add_entry(
        0,
        pipe,
        &match,
        &actions,
        &monitor,
        NULL,
        0,
        status,
        &entry_mac
    );
    /**
        ...
    **/
    return DOCA_SUCCESS;
}
```

In diesem wichtigen Codeabschnitt findet der tatsächliche MAC-Adressen-Rewrite statt. Zunächst wird nun der explizite Filterteil auf der Ebene eines einzelnen Entry gesetzt. Dazu erhält die Match-Datenstruktur ein IP-Bitfeld, bei dem nur das letzte Bit gesetzt ist. Das heißt, dieses Entry fängt alle Pakete ab, deren Source-IP-Adresse auf dem letzten Bit eine 1 gesetzt hat. Daraufhin muss die Action ID definiert werden. Diese Action ID wird vor allem zum späteren Logging verwendet, um zu überprüfen, wie viele Pakete von dieser Action verarbeitet wurden. Anschließend sind die entsprechenden MAC-Adressen in der

`actions.outer.eth.dst_mac` und der `actions.outer.src_mac` angegeben und werden in diesem Schritt entsprechend modifiziert. Mittels `doca_flow_pipe_add_entry()` kann nun das Entry der entsprechenden Pipe zugeordnet werden. Das im Code gezeigte Entry ist lediglich eines für ein entsprechendes Backend, dessen MAC-Adresse in der dst_mac angegeben wird. Sollen nun mehr Backends hinzugefügt werden, benötigt es eine abermalige Ausführung.

6.2.4 Logging und Datenerfassung

Der XenoFlow-Lastverteiler hat eine Echtzeit-Datenerfassung mit implementiert. Dazu wird in einem periodischen Intervall ein Funktionsaufruf gemacht:

```
doca_flow_shared_resources_query(
    DOCA_FLOW_SHARED_RESOURCE_COUNTER,
    shared_counter_ids,
    query_results_array,
    nb_ports
);
```

Dessen Ergebnis ist eine Datenstruktur, die einfach ausgelesen werden kann, um die entsprechenden Werte zur Laufzeit auf der Kommandozeile ausgeben zu können.

6.2.5 Bemerkung

Alle Funktionalitäten wurden aus dem von NVidia bereitgestellten Headerfile `doca_flow.h` entnommen. Die tatsächlichen Implementationen sind **nicht** quelloffen. Es ist nicht ersichtlich, welche Algorithmen zur Anwendung kamen. Die gegebene Headerdatei ist zwar kommentiert, es wäre allerdings deutlich einfacher, das Projekt quelloffen zu halten, sodass ein genauerer Einblick in die interne Funktionsweise ermöglicht wird. Die Libraries werden als .so-Dateien bereitgestellt.

7 Messungen und Experimente

Im Verlauf der Entwicklung des XenoFlow Lastverteilers sind immer wieder Behauptungen seitens NVidia aufgetaucht, die sehr vielversprechend klangen. Um die Leistung der BlueField-3 jedoch genauer beschreiben zu können und zu überprüfen, ob die Behauptungen so auch reproduzierbar sind, wurden mehrere Experimente auf der Karte durchgeführt und diverse Messungen gemacht. Im folgenden Abschnitt werden die Ergebnisse und Messaufbauten vorgestellt.

7.1 Forschungsfragen

In Zusammenarbeit mit der Arbeitsgruppe wurden 3 Fragen in den Vordergrund gestellt. Alle diese 3 sind direkt dem Marketingmaterial von NVidia entnommen und sollen sicherstellen, dass die Karte in möglichst vielen Situationen die gewünschte und proklamierte Leistung erreicht.

- Ist das Versprechen von Zero-Overhead-Verarbeitung ohne Paketverlust aus der NVidia Dokumentation so wahrheitsgetreu?
- Ist es tatsächlich möglich, auch unabhängig von der Paketgröße, immer die angegebenen 400 Gbit/s zu verarbeiten?
- Inwiefern beeinflusst die Last des Load-Balancers die Round-Trip-Time, also somit die Latenz?

7.2 Messaufbau

Um die Leistung eines Lastverteilers zu quantifizieren, sollte ein möglichst realistisches Szenario für Messungen verwendet werden. Ein denkbares und für diese Arbeit verwendetes Szenario wäre ein Lastverteiler für eingehende UDP-DNS-Requests. Ein DNS-Server stellt normalerweise IP-Adressen bereit, die auf einen entsprechenden Server zeigen. Diese IP-Adresse ist mit einer entsprechenden Domain verbunden. Möchte nun also ein Client auf einen bestimmten Server unter einer Domain zugreifen, so stellt er einen DNS-Request an einen DNS-Server. Da auf diesen Server nun für jede Domain, die aufgelöst werden soll, ein Request eintrifft, ist mit einer Menge von Clients schnell die entsprechende maximale Leistungsfähigkeit des DNS-Servers erreicht. Hier setzen wir nun einen Lastverteiler ein, damit die entsprechenden Requests nicht von einem einzelnen Server beantwortet werden, sondern von einer Reihe von Servern, die jeweils die gleichen Daten an Domain-IP-Adressen-Kombinationen hinterlegt haben. Dieser Lastverteiler ist in unserem Fall der XenoFlow. Dazu sind in einem eigens dafür abgestellten Testnetzwerk fünf Knoten verbunden. Vier

dieser Knoten besitzen eine entsprechende Infiniband-Netzwerkkarte, die es ihnen erlaubt, mit 100 Gbit/s Netzwerkverkehr zu verarbeiten. In dem fünften Knoten **fips-5** ist die BlueField-3 verbaut. Damit die entsprechenden richtigen Backends angesprochen werden können, wurden auf dem Switch, der die Server miteinander verbindet, die entsprechenden MAC-Adressen an die Hardwareports per Konfiguration gebunden.

7.2.1 Knoten

Folgende Knoten mit ihrer entsprechenden Funktion kamen zum Einsatz:

- **fips1** - Stellt den Testclient dar, der im folgenden UDP-DNS-Requests an den Server sendet.
- **fips2** - Backend Server, auf dem sowohl Grundlast ankommt als auch der DNS-Server läuft.
- **fips3** - Backend Server, auf dem nur Grundlast ankommt.
- **fips4** - Lasterzeugender Server
- **fips5** - XenoFlow Knoten, in dem die BlueField-3 verbaut ist.

Die fips* Knoten sind mit jeweils zwei Intel Xeon Silver 4514Y Prozessoren ausgestattet. Diese besitzen jeweils 16 Kerne, womit ein Knoten auf 32 Kerne mit 64 Threads kommt. Die Prozessoren des Hosts besitzen einen Basistakt von 2 GHz und Turbotakt von 3.4 GHz. Außerdem besitzen die Knoten 128 GB DDR4 Arbeitsspeicher. Sie sind mittels Mellanox ConnectX Kabeln verbunden. Die BlueField-3 hat einen ARM-Cortex-A78AE mit 16 Kernen verbaut. Dieser hat einen Takt von 2.4 GHz. Die BlueField-3 hat außerdem 32 GB DDR5 Arbeitsspeicher verbaut. Zusätzlich besitzt die Karte einen Co-Prozessor (Data Path Accelerator) der RISC-V Architektur. Genauer kommt dieser auf einen Basistakt von 1.8 GHz. Letzterer ist vor allem für diese Messreihe relevant.

7.2.2 Trafficgenerator

Um Traffic von einem einzelnen Knoten zu generieren, der möglichst nahe an realistischem Netzwerkverkehr liegt, kam im Zuge dieser Arbeit der T-Rex-Trafficgenerator zum Einsatz. Dieser ist vollständig Open Source. TRex verwendet direkt DPDK, was es ihm ermöglicht, eine sehr große Anzahl von Netzwerkverkehr zu erzeugen, ohne einen großen Overhead auf dem entsprechenden lastgenerierenden System zu erzeugen. Es werden diverse Protokolle unterstützt, wie ARP, IPv6, TCP/UDP und DNS. In den folgenden Versuchsreihen wurde die Grundlast durch ein entsprechendes Pythonskript erzeugt, welches direkt mit der TRex API arbeitet. Das Skript befindet sich ebenfalls im Repository dieser Arbeit.

7.2.3 DNS-Server

Damit Messungen wie die Latenz oder der Paketverlust durchgeführt werden können, bedarf es eines kleinen DNS-Servers, der die Client-Requests beantwortet. Dazu wurde abermals ein kleiner Server in Python implementiert, der auf ein Request für eine Domain-

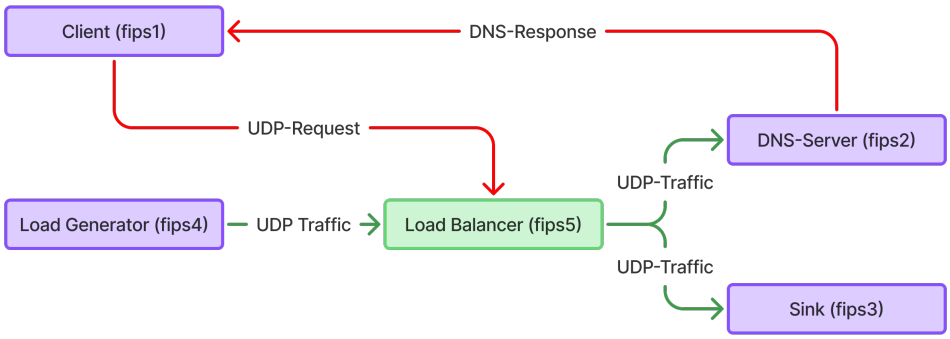

Abbildung 7.1: Latenz- und Paketloss-Aufbau

Adresse mit der zugehörigen IP-Adresse antwortet. Auch hierbei ist der entsprechende Serverquellcode im Repository der Arbeit hinterlegt.

7.3 Messungen

Der XenoFlow LoadBalancer ist aktuell in der Lage, UDP-Pakete, die an ihn per MAC-Adresse gerichtet sind, zu einer Reihe von zugeordneten Backends weiterzuleiten. Folgende Metriken sollen dabei gemessen werden:

- Packet Rate (PPS - Packets per Second)
- Latenz (RTT - Round Trip Time)
- Packet Loss (%)

Zur Messung der Packet Rate und des Packet Losses wird ethtool verwendet, da es so möglich ist, Hardwarestatistiken beziehen zu können, die direkt von der Hardware bereitgestellt werden und somit nicht erst vom Linux-Kernel verarbeitet werden müssen.

7.3.1 Packet Rate

Bei der Paketrate wird in Abhängigkeit von der Paketgröße gemessen, ob sich der maximale mögliche Paketdurchsatz ändert. Dieser Durchsatz wird hierbei in Paketen pro Sekunde gemessen. Die Variable, die verändert wird, ist hierbei die Paketgröße. Somit erhalten wir daraufhin einen Plot, in dem an der Abszisse die Paketgrößen aufgetragen sind und an der Ordinate die Anzahl an maximal möglichen Paketen pro Sekunde. Zusätzlich zu den abgebildeten Paketgrößen wurde eine Basismessung durchgeführt, bei der gar keine Last auf dem Load-Balancer wirkt (siehe Abbildung 7.2).

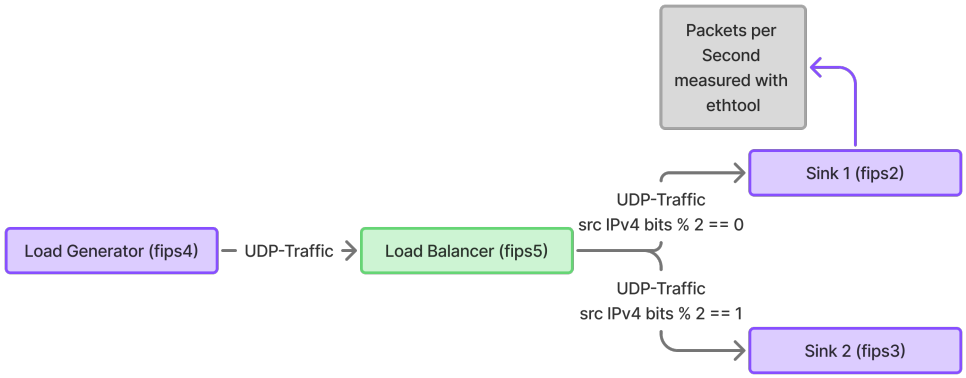

Abbildung 7.2: Paketdurchsatzmessungs-Aufbau

7.3.2 Latenz

Bei der Latenzmessung wird der Laufzeitzuwachs durch die Verarbeitung des XenoFlow LoadBalancers gemessen. Die Messgröße ist hierbei eine Zeiteinheit in Millisekunden oder Mikrosekunden. Die Variable, die verändert wird, ist abermals die Paketgröße, um sicherzustellen, dass die Hardwarebeschleunigung tatsächlich auf der Ebene des Bitfeldes rechnet. Dazu wird in der Auswertung dann ein Plot erstellt, in dem Paketgröße und Latenzzuwachs aufgetragen sind (siehe Abbildung 7.1). Hauptsächlich wird allerdings die wirkende Basislast auf XenoFlow varriert.

7.3.3 Packet Loss

Der Packet Loss misst, ab welcher Last und vor allem wie viele Pakete bei einer bestimmten Last auf dem LoadBalancer verloren gehen. Die gemessene Metrik ist hierbei ein Prozentsatz, der angibt, wie viel anteilig vom tatsächlichen Traffic auch vom Load-Balancer verarbeitet werden kann. Die Variable, die hierbei variiert wird, ist die Last auf den Load-Balancer. Dabei wird die Anzahl der Pakete pro Sekunde Basislast geändert. In der Auswertung wird ähnlich wie bei der Latenz der Grundtraffic dem Packet Loss gegenübergestellt.

7.4 Ergebnisse

Zunächst war das formulierte Ziel, eine vollständige Hardwarebeschleunigung seitens der BlueField-3 verwenden zu können, um den Traffic zu verarbeiten. Dies war leider zunächst so nicht möglich. Die ersten Versuche zeigten durch die sichtbare CPU-Last auf dem BlueField-System deutlich, dass der Netzwerkverkehr nicht vollständig über die Hardwareeinheiten geleitet wurde. Das war doch sehr verwunderlich, da in keinem uns bekannten

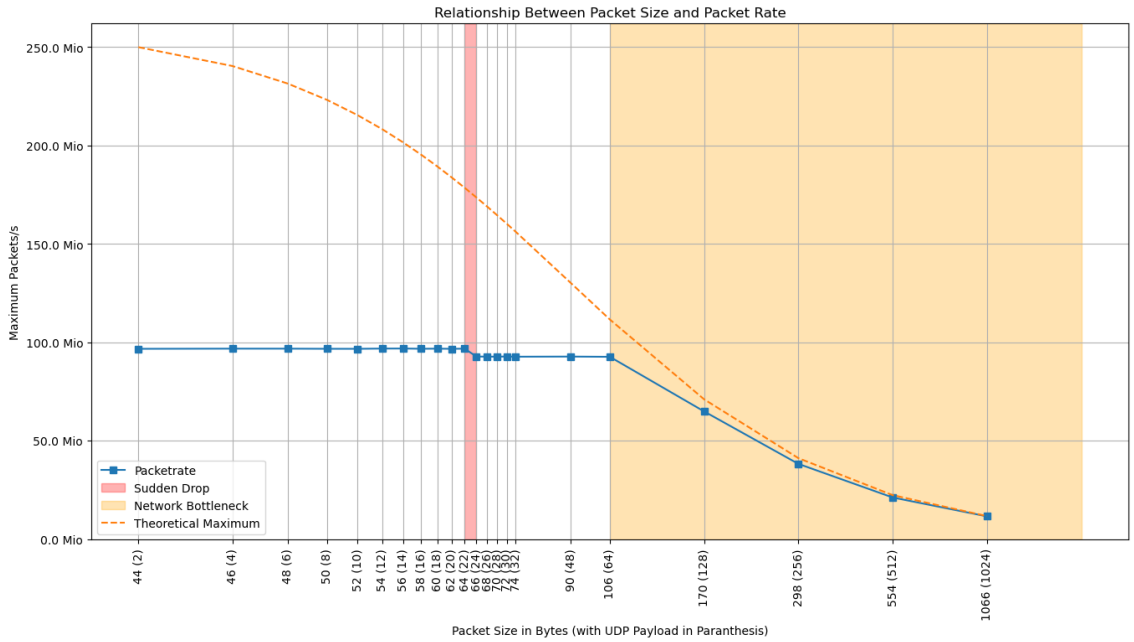

Abbildung 7.3: Pakete pro Sekunde - variierende Paketgröße

Abschnitt der Dokumentation von einer speziellen Konfiguration oder Ähnlichem die Rede war, die diese Funktion aktiviert. Nach einer Weile und vielen Versuchen entstand allerdings die Idee, mittels Open vSwitch den Netzwerkverkehr derart umzuleiten, dass alle Pakete, die auf **Port 0** eintreffen, direkt auf **Port 1** weitergeleitet werden. Nachdem diese Konfiguration mittels des Open vSwitch CLI-Tools vorgenommen wurde, verschwand die Last auf den ARM-Kernen der BlueField sofort. Wurde nun der Lastverteiler gestartet, so war ebenfalls keine Last mehr auf den Kernen zu beobachten. Somit liegt die Vermutung nahe, dass nun der Netzwerkverkehr auf der ASIC-Hardware verarbeitet wurde. Außerdem werden so Mutmaßungen über die Architektur der BlueField-3 möglich. Es ist sehr wahrscheinlich, dass DOCA-Flow-Applikationen wirklich auf der ASIC-Hardware ausgeführt werden und somit ein Off-Path-Switching umsetzen, da weder auf Port-0 noch Port-1 auf dem Betriebssystem der BlueField-3 Pakete beobachtet werden können.

7.4.1 Packet Rate

Bei dem Versuch der Packet Rate wurde gemessen, wie viele Pakete pro Sekunde von der BlueField-3 verarbeitet werden können. Dabei wurde die Paketgröße variiert, indem die Größe der UDP-Payload geändert wird. Sollte nun die Größe der Pakete des Netzwerkverkehrs einen Einfluss darauf haben, wie viele Pakete verarbeitet werden können, so würde sich hier eine Kurve entsprechend abzeichnen. Dies ist so nicht der Fall (siehe Abbildung 7.3). Gemessen wurden UDP-Payloadgrößen von 2 bis 1024 Bytes. Zunächst ist zwischen 2 und 22 Bytes keine Veränderung der möglichen Paketrate sichtbar. Es werden konstant ca. 98 Millionen Pakete pro Sekunde vom Loadbalancer verarbeitet und können auf dem entsprechenden Backend beobachtet werden, auf dem die Pakete ankommen.

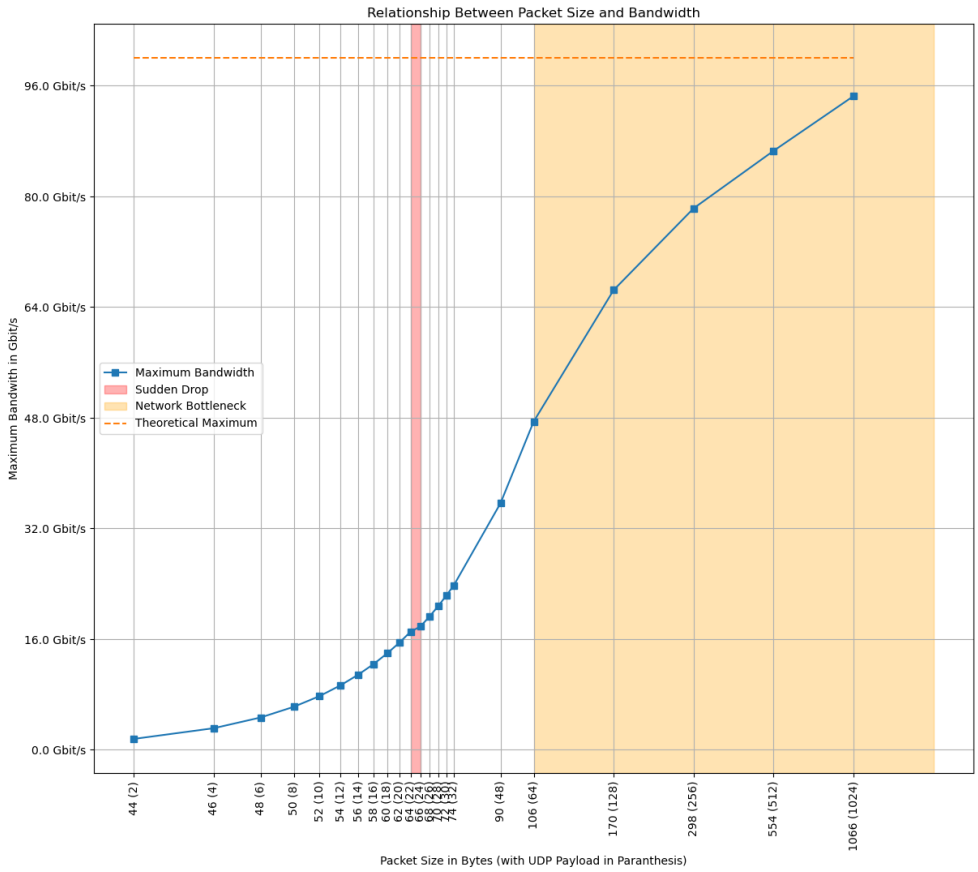

Abbildung 7.4: Pakete pro Sekunde - verfügbare Bandbreite

Ethernet- und IP-Header machen zusammen 42 Bytes aus. Sind nun aber die Pakete mit einer UDP-Payload von 24 Bytes versehen, also insgesamt 64 Bytes Paketgröße, so bricht die Verarbeitungsrate auf ca. 92 Millionen Pakete pro Sekunde ein. Erklärbar ist dieser Effekt vermutlich damit, dass durch die feste Breite von den Schnittstellen der ASIC-Hardware eine andere Route durch die Beschleuniger genommen wird, da diese Pakete als große Pakete behandelt werden.

Klar davon zu trennen ist die Tatsache, dass in dem gegebenen Messbett eine maximale Bandbreite von 100 Gbit/s zur Verfügung stand. Dementsprechend war es nicht möglich, größere UDP-Payloads als 64 Bytes zu messen. Ab diesem Punkt ist der Flaschenhals das Netzwerk, weswegen dieser Bereich in Abbildung 7.3 mit Orange gekennzeichnet wurde. Bei Betrachtung von Abbildung 7.4, in der das Verhältnis zwischen der Paketgröße und der maximalen möglichen Bandbreite abgetragen ist, fällt sofort auf, dass die Erklärung seitens NVidia von angeblichen 400 Gbit/s so nicht nachgewiesen werden konnte. Dies ist vor allem der Tatsache geschuldet, dass in dieser Arbeit vor allem kleine Pakete gemessen wurden.

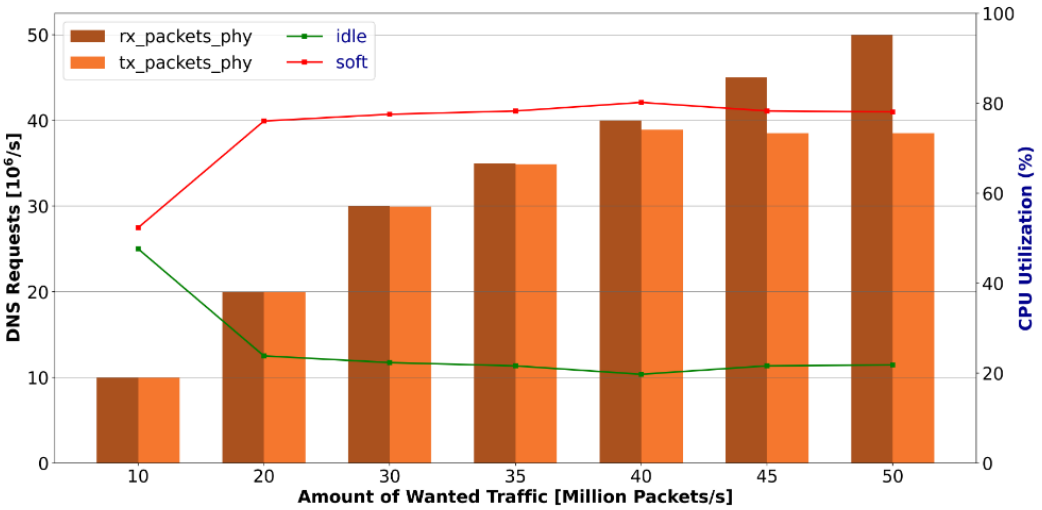

Abbildung 7.5: Katran Lastverteiler Leistung bei 254 Quell IP Adressen [32]

So können mit Paketgrößen von 64 Bytes, was eine sehr typische Paketgröße für DNS-Pakete ist, gerade einmal 16 Gbit/s erreicht werden. Dies ist in etwa 4 % der beworbenen Leistung und somit weit unter der erwarteten Performance. Vergleichen wir allerdings die Leistung zu der eines gängigen Software-Lastverteilers wie Katran in der Arbeit von Phillip Ungrund, so sehen wir eine ungefähr doppelt so hohe Menge von maximalen Paketen pro Sekunde, die verarbeitet werden können [32] (Abbildung 7.5).

In Abbildung 7.6 ist ein Vergleich zwischen den unterschiedlichen Lastverteilern hinsichtlich ihrer maximalen Pakete/Sekunde Verarbeitungs-Geschwindigkeit. Kakao Corporation ist ein südkoreanischer Cloud-Anbieter, der einen ähnlichen Ansatz verfolgt hat. Es wurde bei deren Versuch ein eBPF-Lastverteiler, ähnlich wie Katran, implementiert [40]. Dazu wurden aus ihren Messungen zwei entnommen. Im Loopback-Versuch haben sie den eingehenden Traffic eins zu eins wieder ausgegeben und wollten so eine Baseline der maximal möglichen Leistung schaffen. Bei dem 255-Backends-Versuch gab es einen VIP-Server, auf dem die eigentliche Lastverteilung mittels eBPF stattfand. Beide Ansätze sind zwar langsamer als Katran, aber befinden sich bezüglich der maximalen Leistung nahe beieinander. XenoFlow jedoch kann mit einem Speed-Up von über zwei gegenüber jedem der anderen Lastverteiler gemessen werden.

7.4.2 Latenz

Die Latenz wurde gemessen, indem zunächst eine Grundlast auf den Lastverteiler ausgeübt wurde. Nun wurde von fips-1 ein einzelner DNS-Request an den Lastverteiler geschickt, der diesen Request aufgrund seiner Source-IP an fips-2 weitergeleitet hat. Fips-2 hat daraufhin eine Antwort an fips-1 mit der IP gesendet. Auf fips-1 wurde dabei die Zeit gemessen, die das Paket für diesen kompletten Pfad gebraucht hat. Dieser Wert wird auch als Round-Trip-Time, kurz RTT, bezeichnet. In dieser Versuchsreihe wurde nun variiert, wie viel des Grundtraffics an die beiden Backends geschickt wird. Dies wird anhand der Source-IP-

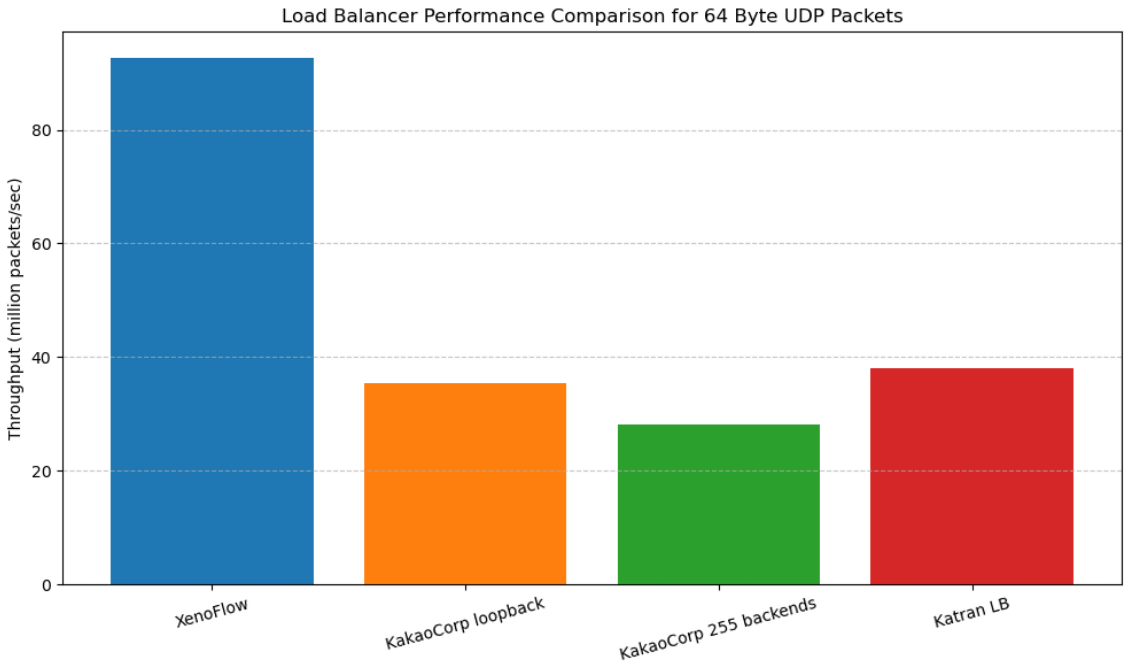

Abbildung 7.6: Vergleich von unterschiedlichen Lastverteilern vs. XenoFlow [40][32]

Felder gesteuert. Somit kann beobachtet werden, ob es Algorithmen gibt, die den Traffic stochastisch verteilen, oder ob es tatsächlich eine Paketverarbeitung für jedes einzelne Paket gibt.

Single Endpoint

Bei dem Versuch aus Abbildung 7.7 wird die gesamte Grundlast an den gleichen Server weitergeleitet, auf dem auch der DNS-Server liegt. Hierbei ist der Latenzzuwachs vermutlich durch die Last auf dem Backendsystem zu begründen. Der DNS-Server muss sich mit den restlichen kernelinternen Prozessen um CPU-Zeit streiten, was sich in teilweise schlechteren Antwortzeiten bemerkbar macht. Allerdings ist der Latenzzuwachs sehr gering.

50/50

Es ist in Abbildung 7.7 bei einer 50/50-Lastverteilung kein signifikanter Latenzzuwachs in Abhängigkeit von der Basislast zu erkennen.

70/30

Es ist in Abbildung 7.7 bei einer 70/30-Lastverteilung kein signifikanter Latenzzuwachs in Abhängigkeit von der Basislast zu erkennen.

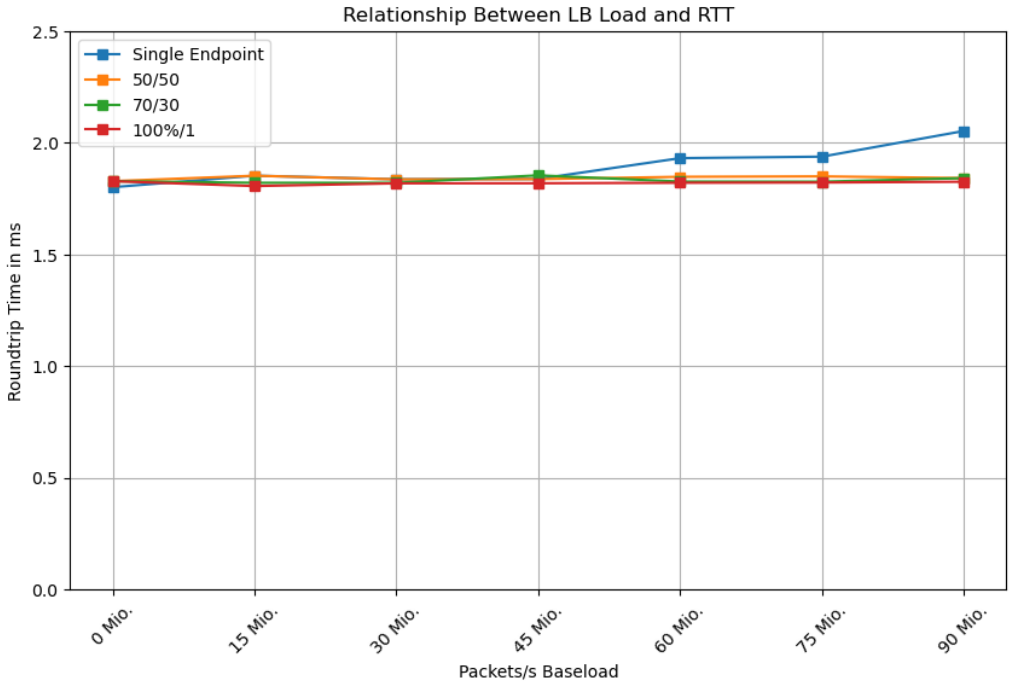

Abbildung 7.7: Latenzmessungen in unterschiedlichen Szenarien

Worstcase

Bei dem Versuch wird der komplette Netzwerkverkehr außer dem DNS-Request-Paket an das Backend ohne DNS-Server weitergeleitet. Gäbe es einen Zusammenhang mit dem Matching der Pakete und der entsprechenden Verteilung, so würde es bei diesem Versuch vermutlich am ehesten sichtbar werden. Es ist in Abbildung 7.7 kein signifikanter Latenzzuwachs in Abhängigkeit von der Basislast zu erkennen.

7.4.3 Paketverlust

In Abbildung 7.8 ist erwartungsgemäß ein lineares Wachstum des Paketverlustes ab der entsprechenden maximalen Verarbeitungsgeschwindigkeit von ca. 95 Millionen Paketen pro Sekunde zu beobachten. Für den entsprechenden Versuch wurden Pakete der Größe 64 Byte verwendet. Somit ist der Paketverlust direkt proportional zu dem im Versuch der Bandbreite ermittelten maximalen Paketdurchsatz.

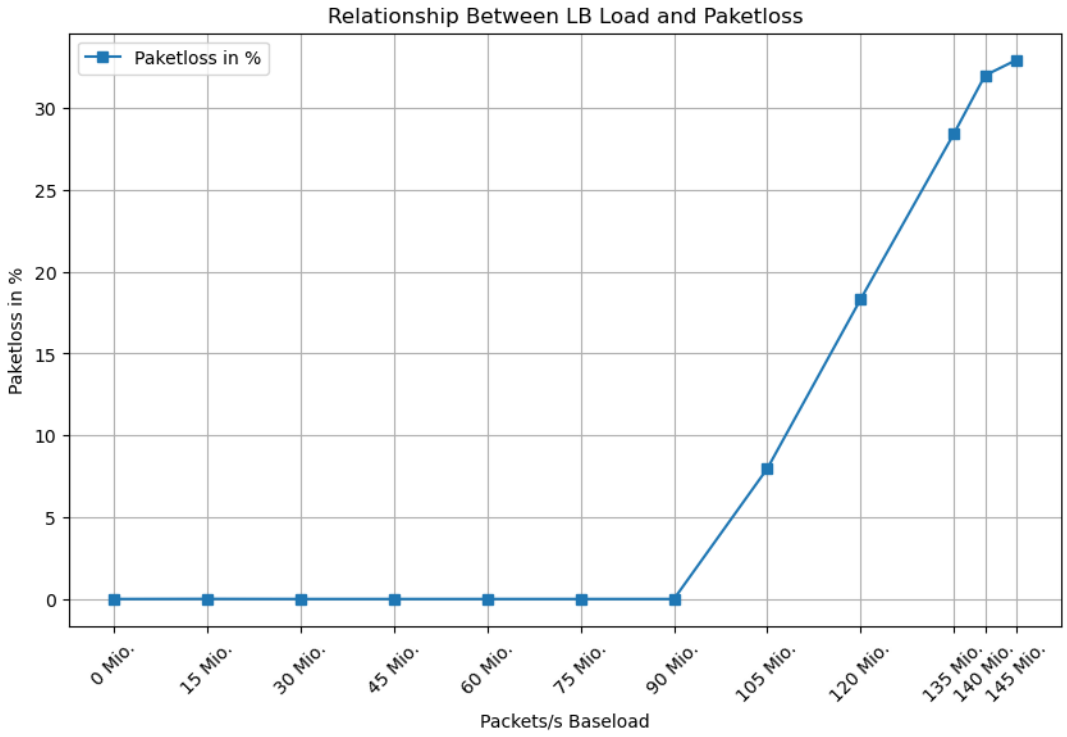

Abbildung 7.8: Paketverlust vs. Pakete pro Sekunde Basislast

8 Fazit

Ziel dieser Arbeit war die Entwicklung und Implementierung sowie die anschließende Evaluation eines Lastverteilers auf einer DPU. Dazu wurde die BlueField-3 als Plattform gewählt. Es wurden unterschiedliche Bereiche der Hardwarearchitektur von NVIDIA analysiert und dazu relevante Forschungsfragen formuliert. Es konnte im Vergleich zu anderen Softwarelastverteilern eine erheblich höhere Durchsatzrate gezeigt werden. Der Lastverteiler ist ein Proof-of-Concept, welches beweist, dass es durchaus möglich ist, Anwendungen sinnvoll auf einer Smart NIC zu implementieren. Allerdings ist dies nur mit einer tieferen Analyse der Schnittstellen und der Hardware möglich, um innerhalb der Implementierung von den gegebenen Hardwarebeschleunigern Gebrauch machen zu können und eventuell auftretenden Bottlenecks aus dem Wege zu gehen.

8.1 Versprechen der Zero-Overhead-Verarbeitung

Das Versprechen seitens NVIDIA, dass der vollständige Netzwerkverkehr ohne Overhead und vor allem ohne Paketverlust verarbeitet werden kann, ließ sich so nicht reproduzieren. Es war in den Versuchen, bei denen Last auf den Lastverteiler XenoFlow ausgeübt wurde, keine Veränderung der Latenz messbar. Allerdings wurde bei einer Paketrate von ca. 95 - 98 Millionen Paketen pro Sekunde nicht mehr der komplette Datenverkehr verarbeitet. Es kam zu erheblichen Paketverlusten. Somit ist die Behauptung seitens NVIDIA als nicht allgemein gültig zu betrachten. Positiv festzustellen ist allerdings, dass die Paketverarbeitungsrate nicht von der Paketgröße abhängt.

8.2 Einfluss der Paketgröße auf den Datendurchsatz

Die beworbenen 400 Gbit/s Datendurchsatz konnten in unseren Experimenten nicht erreicht werden. Dazu sei erwähnt, dass das verwendete Netzwerk aufgrund der in den Testknoten verbauten Netzwerkkarten einen Durchsatz von 100 Gbit/s hatte. Aber auch dieser Wert ließ sich nur ab einer Paketgröße von über 1024 Bytes erreichen. Dabei handelt es sich um eine Paketgröße, welche im Kontext der Verwendung mit den meisten Anwendungen eher untypisch ist. Klassische Paketgrößen kommen selbst bei einer vollständigen Hardwareauslagerung auf eine Bandbreite von nicht mehr als 64 Gbit/s (siehe Abbildung 7.4).

8.3 Einfluss der Last auf die Round-Trip-Time

Es konnte nachgewiesen werden, dass die Auslastung der Hardwareports sowie der DPA-Architektur keinen messbaren Einfluss auf die Verarbeitungslatenz sowie Round-Trip-Time

hat. Dazu wurde der Vergleich von mehreren Lastverteilungsszenarien durchgeführt. Somit konnte diese Behauptung des Herstellers als richtig erwiesen werden. Außerdem war es in dem Zuge, zwar mit Hürden, möglich, vollständigen Gebrauch der angepriesenen Hardwarebeschleuniger auf der BlueField-3 zu machen.

8.4 Ausblick

Lastverteiler in Rechenzentren werden auch in den nächsten Jahrzehnten mit steigendem Interesse verfolgt werden. So zeigen etwaige Cloud-Anbieter wie Amazon, Google oder Microsoft sehr großes Interesse an Lastverteilungslösungen. Der XenoFlow-Lastverteiler ist zum Zeitpunkt der Abgabe dieser Arbeit in der Lage, UDP-Pakete auf eine Anzahl von fest definierten Servern anhand ihrer IP- und MAC-Adresse zu verteilen. Dabei ist derzeit insbesondere keine dynamische Anpassung der verfügbaren Endpunkte möglich. Eine interessante Weiterentwicklung könnte es also sein, XenoFlow um einen Mechanismus zu erweitern, der es ihm ermöglicht, die Backends, an die weitergeleitet werden soll, anzupassen. Dazu wäre eine weitere Kommunikationsebene denkbar, auf der XenoFlow mit den Endpunkten kommunizieren kann. Somit wären die Endpunkte in der Lage, ihren aktuellen Zustand mit XenoFlow abzustimmen, damit dieser intelligentere Lastverteilungsentscheidungen fällen könnte. Dazu wäre ein Algorithmus denkbar, der dynamisch in Erfahrung bringt, ob der Quell-IP-Adresse bereits ein Backend zugeordnet wurde. Ist dies nicht der Fall, so wird ein neues Backend gewählt. Es könnte für jedes Backend eine eigene Pipe oder ein entsprechendes Entry angelegt werden. Welche dieser Arten die schnellere und effizientere ist, wäre abermals eine spannende Frage für kommende Arbeiten. Diese Art der Lastverteilung wäre deutlich dynamischer im Vergleich zum aktuellen statischen Klassifizieren von IP-Adressen in XenoFlow.

Außerdem wäre es interessant, XenoFlow um das Protokoll TCP zu erweitern. TCP wird von wesentlich mehr Applikationen verwendet. Dazu müsste allerdings eine Art und Weise entwickelt werden, wie mit offenen Verbindungen o. ä. auf dem XenoFlow-Lastverteiler gearbeitet wird. Zudem wäre ein Einsatz in Edge-Computing oder Kubernetes-Clustern denkbar und bietet daher verschiedene Ansätze, wie XenoFlow zum Einsatz kommen kann.

Unabhängig von XenoFlow können aber auch andere Anwendungen auf der DPU implementiert werden. Es konnte eine starke Leistungssteigerung mittels ASIC-Verwendung gezeigt werden, was die Hoffnung bestärkt, dass auch andere Programme davon profitieren können. Energieeffizientes Rechnen ist ein hochaktuelles Thema und wird auch in den kommenden Jahrzehnten eine maßgebliche Thematik der Rechenzentren und somit der Informatik bleiben.

Abbildungsverzeichnis

A Source Files

The source files and the corresponding repository can be accessed by contacting the second supervisor: Max Schrötter.

Literatur

[1] Andrew S. Tanenbaum und Herbert Bos. *Modern Operating Systems*. Pearson Education, 2007 (siehe S. 1).

[2] Luiz André Barroso und Urs Hölzle. „The case for energy-proportional computing“. In: *Computer* 40.12 (2007), S. 33–37 (siehe S. 1).

[3] Norman P. Jouppi u. a. „In-datacenter performance analysis of a tensor processing unit“. In: *ACM/IEEE International Symposium on Computer Architecture (ISCA)* (2017), S. 1–12 (siehe S. 1).

[4] Elie F. Kfoury u. a. „A Comprehensive Survey on SmartNICs: Architectures, Development Models, Applications, and Research Directions“. In: *IEEE Access* 12 (2024), S. 107297–107336 (siehe S. 2).

[5] Seo Jin Park u. a. „Lovelock: Towards Smart NIC-Hosted Clusters“. In: *SIGENERGY Energy Inform. Rev.* 4.5 (Apr. 2025), S. 172–179. URL: https://doi.org/10.1145/3727200.3727226 (siehe S. 2).

[6] Nuha Alshuqayran, Nour Ali und Roger Evans. „A Systematic Mapping Study in Microservice Architecture“. In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. 2016, S. 44–51 (siehe S. 2).

[7] Charles E Spurgeon und Joann Zimmerman. *Ethernet switches: An introduction to network design with switches*. Ö'Reilly Media, Inc.", 2013 (siehe S. 3).

[8] Thomas N. Theis und H.-S. Philip Wong. „The End of Moore's Law: A New Beginning for Information Technology“. In: *Computing in Science Engineering* 19.2 (2017), S. 41–50 (siehe S. 2).

[9] Georgios P Katsikas u. a. „What you need to know about (smart) network interface cards“. In: *International Conference on Passive and Active Network Measurement*. Springer. 2021, S. 319–336 (siehe S. 3).

[10] Xingda Wei u. a. „Characterizing off-path {SmartNIC} for accelerating distributed systems“. In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 2023, S. 987–1004 (siehe S. 4, 5).

[11] Norman Einspruch. *Application specific integrated circuit (ASIC) technology*. Bd. 23. Academic Press, 2012 (siehe S. 6).

[12] D.L. Eager, J. Zahorjan und E.D. Lazowska. „Speedup versus efficiency in parallel systems“. In: *IEEE Transactions on Computers* 38.3 (1989), S. 408–423 (siehe S. 7).

[13] Lasse Thostrup u. a. „A DBMS-centric Evaluation of BlueField DPUs on Fast Networks.“ In: *ADMS@ VLDB* 2022 (2022) (siehe S. 12, 13).

[14] Benjamin Michalowicz u. a. „Battle of the bluefields: An in-depth comparison of the bluefield-2 and bluefield-3 smartnics“. In: *2023 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE. 2023, S. 41–48 (siehe S. 14).

[15] NVIDIA Corporation. *NVIDIA BlueField Data Processing Unit*. https://www.nvidia.com/en-us/networking/products/data-processing-unit/. Zugriff am 12. Mai 2025. 2025 (siehe S. 14).

[16] NVIDIA Corporation. *DPA Subsystem Programming Guide*. Zugriff am 12. Mai 2025. 2025. URL: https://docs.nvidia.com/doca/sdk/dpa+subsystem/index.html (siehe S. 16).

[17] NVIDIA Corporation. *NVIDIA DOCA Software Framework*. https://developer.nvidia.com/networking/doca. Zugriff am 12. Mai 2025. 2025 (siehe S. 16, 17).

[18] NVIDIA Corporation. *DOCA Flow Programming Guide*. Zugriff am 12. Mai 2025. 2025. URL: https://docs.nvidia.com/doca/sdk/doca+flow/index.html (siehe S. 18).

[19] NVIDIA Corporation. *DOCA Flow Programming Guide (Version 1.2)*. Zugriff am 12. Mai 2025. 2025. URL: https://docs.nvidia.com/doca/archive/doca-v1.2/flow-programming-guide/index.html (siehe S. 19–21).

[20] Rahul Soni. „Load Balancing with Nginx“. In: *Nginx: From Beginner to Pro*. Springer, 2016, S. 153–171 (siehe S. 27).

[21] Yossi Azar. „On-line load balancing“. In: *Online algorithms: the state of the art* (2005), S. 178–195 (siehe S. 25).

[22] Young Sik Hong, JH No und SY Kim. „DNS-based load balancing in distributed Web-server systems“. In: *The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06)*. IEEE. 2006, 4–pp (siehe S. 25).

[23] Tony Bourke. *Server load balancing*. Ö'Reilly Media, Inc.", 2001 (siehe S. 26).

[24] Ali Akbar Neghabi u. a. „Load balancing mechanisms in the software defined networks: a systematic and comprehensive review of the literature“. In: *IEEE access* 6 (2018), S. 14159–14178 (siehe S. 27).

[25] Indrani Vasireddy, G Ramya und Prathima Kandi. „Kubernetes and docker load balancing: State-of-the-art techniques and challenges“. In: *International Journal of Innovative Research in Engineering and Management* 10.6 (2023), S. 49–54 (siehe S. 28).

[26] Adi Fu. *A Future of AI through the Semiconductor Looking Glass*. https://medium.com/@adi.fu7/a-future-of-ai-through-the-semiconductor-looking-glass-ca24451517ae. Accessed: 2025-05-14. 2024 (siehe S. 7).

[27] Byeongho Kim u. a. „The breakthrough memory solutions for improved performance on llm inference“. In: *IEEE Micro* 44.3 (2024), S. 40–48 (siehe S. 6).

[28] Or Gerlitz und Andy Gospodarek. „Taking Control of your SmartNIC“. In: *Netdev 0x14 Conference*. Accessed: 2025-05-14. Aug. 2020. URL: https://netdevconf.info/0x14/pub/slides/39/Netdev%200x14%20--%20Taking%20Control%20of%20your%20SmartNIC%20v1.pdf (siehe S. 5).

[29] Qurren. *Buffalo IFC-ILP4 VIA VT6306*. https://commons.wikimedia.org/wiki/File:Buffalo_IFC-ILP4_VIA_VT6306.jpg. Accessed: 2025-05-14. 2007 (siehe S. 5).

[30] eBay seller: electronicssolutions. *Nvidia BlueField 1 MBF1L516A-CSCAT ETH 100GbE 2x QSFP28 Gen3.0/4.0 x8 Crypto enabled*. https://www.ebay.com/itm/235308508672. Accessed: 2025-05-14. 2025 (siehe S. 13).

[31] Xuzheng Chen u. a. „Demystifying datapath accelerator enhanced off-path smartnic“. In: *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*. IEEE. 2024, S. 1–12 (siehe S. 8).

[32] Phillip Ungrund. „Design and Integration of Shared Memory IPC for Fast Intrusion Prevention in a Modern Load Balancer“. Magisterarb. Universität Potsdam, 2024 (siehe S. 27, 49, 50).

[33] Jianshen Liu u. a. „Performance characteristics of the bluefield-2 smartnic“. In: *arXiv preprint arXiv:2105.06619* (2021) (siehe S. 9).

[34] Elie F Kfoury u. a. „A comprehensive survey on smartnics: Architectures, development models, applications, and research directions“. In: *IEEE Access* (2024) (siehe S. 9).

[35] Tianyi Cui u. a. „Laconic: Streamlined load balancers for SmartNICs“. In: *arXiv preprint arXiv:2403.11411* (2024) (siehe S. 10).

[36] FiberMall. *Verständnis der NVIDIA BlueField-3 DPU*. de. Zugriff am 28. Mai 2025. 2024. URL: https://www.fibermall.com/de/blog/understand-nvidia-bluefield-3-dpu.htm (siehe S. 15, 28).

[37] Wikimedia Commons. *IP Address Illustration (Ip_pic.jpg)*. en. Zugriff am 28. Mai 2025. n.d. URL: https://commons.wikimedia.org/wiki/File:Ip_pic.jpg (siehe S. 19).

[38] NVIDIA Corporation. *DOCA Flow SDK Documentation – Pipe Matching or Action Applying*. en. Zugriff am 28. Mai 2025. NVIDIA. 2024. URL: https://docs.nvidia.com/doca/sdk/doca+flow/index.html#src-3660567507_id-.DOCAFlowv3.0.0-PipeMatchingorActionApplying (siehe S. 22).

[39] WorldTech IT. *F5 BIG-IP 5000s–5050s – 5250v–5200v Hardware Datasheet*. en. Zugriff am 28. Mai 2025. 2025. URL: https://wtit.com/f5-products/f5-big-ip-5000s-5050s-5250v-5200v-hardware-datasheet/ (siehe S. 27).

[40] Jung-Bok Lee u. a. „High-performance software load balancer for cloud-native architecture“. In: *IEEE Access* 9 (2021), S. 123704–123716 (siehe S. 49, 50).